

PROCEDURES AS A REPRESENTATION FOR DATA
IN A COMPUTER PROGRAM FOR UNDERSTANDING
NATURAL LANGUAGE

by

Terry Winograd

B.A., The Colorado College
(1966)

SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE
DEGREE OF DOCTOR OF
PHILOSOPHY

at the

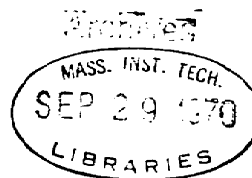
MASSACHUSETTS INSTITUTE OF
TECHNOLOGY

August, 1970

Signature of Author
Department of Mathematics, August 24, 1970

Certified by Thesis Supervisor

Accepted by
Chairman, Departmental Committee
on Graduate Students



PROCEDURES AS A REPRESENTATION FOR DATA
IN A COMPUTER PROGRAM FOR UNDERSTANDING
NATURAL LANGUAGE

by

Terry Winograd

Submitted to the Department of Mathematics on August 24, 1970 in partial fulfillment of the requirements for the degree of Doctor of Philosophy

ABSTRACT

This paper describes a system for the computer understanding of English. The system answers questions, executes commands, and accepts information in normal English dialog. It uses semantic information and context to understand discourse and to disambiguate sentences. It combines a complete syntactic analysis of each sentence with a "heuristic understander" which uses different kinds of information about a sentence, other parts of the discourse, and general information about the world in deciding what the sentence means.

It is based on the belief that a computer cannot deal reasonably with language unless it can "understand" the subject it is discussing. The program is given a detailed model of the knowledge needed by a simple robot having only a hand and an eye. We can give it instructions to manipulate toy objects, interrogate it about the scene, and give it information it will use in deduction. In addition to knowing the properties of toy objects, the program has a simple model of its own mentality. It can remember and discuss its plans and actions as well as carry them out. It enters into a dialog with a person, responding to English sentences with actions and English replies, and asking for clarification when its heuristic programs cannot understand a sentence through use of context and physical knowledge.

In the programs, syntax, semantics and inference are integrated in a "vertical" system in which each part is constantly communicating with the others. We have explored several techniques for integrating the large bodies of complex knowledge needed to understand language. We use Systemic Grammar, a type of syntactic analysis which is designed to deal with semantics. Rather than concentrating on the exact form of rules for the shapes of linguistic constituents, it is structured around choices for conveying meaning. It abstracts the relevant features of the linguistic structures which are important for interpreting their meaning.

We represent many kinds of knowledge in the form of procedures rather than tables of rules or lists of patterns. By developing special procedural languages for grammar, semantics, and deductive logic, we gain the flexibility and power of programming languages while retaining the regularity and understandability of simpler rule forms. Each piece of knowledge can be a procedure, and can call on any other piece of knowledge in the system.

Thesis Supervisor: Seymour A. Papert
Title: Professor of Applied Mathematics

The author would like to express his gratitude to the many people who helped and encouraged him in this work: to the members of the AI group at Project MAC who provided a critical forum for ideas; particularly to Carl Hewitt for introducing him to PLANNER, to Eugene Charniak and Gerald Sussman for their efforts in implementing Micro-Planner, and for many hours of provoking discussions about the problems of natural language and semantics; to Professor Seymour Papert for supervising the thesis, to Professors Marvin Minsky and Michael Fischer for their suggestions and criticisms; to the PDP-10 for patiently and uncomplainingly typing many drafts of the text; and to Carol for always.

TABLE OF CONTENTS

PREFACE -- Talking to Computers	8
1. INTRODUCTION	15
1.1 General Description	15
1.1.1 What is Language?	16
1.1.2 Organization of the Language Process	20
1.2 Implementation of the System	27
1.3 Sample Dialog	32
2. SYNTAX.	53
2.1 Basic Approach to Syntax	53
2.1.1 Syntax and Meaning	53
2.1.2 Parsing	57
2.2 A Description of PROGRAMMAR	60
2.2.1 Grammar and Computers	60
2.2.2 Context-free and Context-sensitive Grammars	62
2.2.3 Systemic Grammar	63
2.2.4 Grammars as Programs	66
2.2.5 The Form of PROGRAMMAR Grammars	68
2.2.6 The Context-sensitive Aspects	71
2.2.7 Ambiguity and Understanding	77
2.2.8 Summary	80
2.3 A Grammar of English	82
2.3.1 About the Grammar	82
2.3.2 Units, Rank, and Features	84
2.3.3 The Clause	93
2.3.4 Noun Groups	110
2.3.5 Preposition Groups	118
2.3.6 Adjective Groups	120
2.3.7 Verb Groups	122
2.3.8 Words	124
2.3.9 Following the Parser in Operation	134
2.4 Programming Details	152

2.4.1	Operation of the System	152
2.4.2	Special Words	154
2.4.3	The Dictionary	157
2.4.4	Backup Facilities	159
2.4.5	Messages	161
2.4.6	The Form of the Parsing Tree	162
2.4.7	Variables Maintained by the System	163
2.4.8	Pointers	164
2.4.9	Feature Manipulating	165
2.5	Comparison with Other Parsers	167
2.5.1	Older Parsers	167
2.5.2	Augmented Transition Networks	169
2.5.3	Networks and Programs	171
3.	INFERENCE	178
3.1	Basic Approach to Meaning	178
3.1.1	Representing Knowledge	178
3.1.2	Philosophical Considerations	182
3.1.3	Complex Information	185
3.1.4	Questions, Statements, and Commands	190
3.2	Comparison with Previous Programs	193
3.2.1	Special Format Systems	193
3.2.2	Text-based Systems	194
3.2.3	Limited Logic Systems	196
3.2.4	General Deductive Systems	200
3.2.5	Procedural Deductive Systems	204
3.3	Programming in PLANNER	211
3.3.1	Basic Operation of PLANNER	211
3.3.2	Backup	215
3.3.3	Other Theorem Provers and Languages	218
3.3.4	Controlling the Data Base	221
3.3.5	Events and States	223
3.3.6	PLANNER Functions	227
3.4	The BLOCKS World	230
3.4.1	Objects	230
3.4.2	Relations	233

3.4.3	Actions	237
3.4.4	Memory	243
4.	SEMANTICS	248
4.1	What is Semantics?	248
4.1.1	The Province of Semantics	248
4.1.2	The Semantic System	250
4.1.3	Words	253
4.1.4	Discourse	257
4.1.5	Ambiguity	261
4.1.6	Goals of a Semantic Theory	265
4.2	Semantic Structures	268
4.2.1	Object Semantic Structures	268
4.2.2	Relative Clauses	276
4.2.3	Preposition Groups	281
4.2.4	Types of Object Descriptions	283
4.2.5	The Meaning of Sentences	288
4.3	The Semantics of Discourse	302
4.3.1	Pronouns	303
4.3.2	Substitutes and Incompletes	308
4.3.3	Overall Discourse Context	311
4.4	Generation of Responses	314
4.4.1	Patterned Responses	314
4.4.2	Answering Questions	317
4.4.3	Naming Objects and Events	323
4.4.4	Generating Discourse	326
4.4.5	Future Development	328
5.	RELATED TOPICS	330
5.1	Teaching, Telling, and Learning	330
5.1.1	Types of Knowledge	330
5.1.2	Syntax	332
5.1.3	Inference	337
5.1.4	Semantics	341
5.2	Directions for Future Research	345

BIBLIOGRAPHY
BIOGRAPHICAL NOTE

351
356

Preface -- Talking to Computers

Computers are being used today to take over many of our jobs. They can perform millions of calculations in a second, handle mountains of data, and perform routine office work much more efficiently and accurately than humans. But when it comes to telling them what to do, they are tyrants. They insist on being spoken to in special computer languages, and act as though they can't even understand a simple English sentence.

Let us envision a new breed of computers which could take instructions in a way suited to their jobs. We will talk to them just like we talk to a research assistant, librarian, or secretary, and they will carry out our commands and provide us with the information we ask for. If our instructions aren't clear enough, they will ask for more information before they do what we want, and this dialog will all be in English.

Why isn't this being done now? Aren't computers translating foreign languages and conducting psychiatric interviews? Surely it must be easier to understand simple requests for information than to understand Russian or a person's psychological problems. The key to this question is in understanding what we mean by "understanding". Computers are very adept at manipulating symbols -- at shuffling around strings of letters and words, looking them up in dictionaries, and rearranging them. In the early days of computing, some people thought that this might be just what was needed to translate languages. The

government supported a tremendous amount of research into language translation, and a number of projects tried different approaches. In 1966 a committee of the National Academy of Sciences wrote a report evaluating this research and announced sadly that it had been a failure. Every project ran up against the same brick wall -- the computer didn't know what it was talking about.

When a human reader sees a sentence, he uses knowledge to understand it. This includes not only grammar, but also his knowledge about words, the context of the sentence, and most important, his knowledge about the subject matter. A computer program supplied with only a grammar for manipulating the symbols of language could not produce a translation of reasonable quality. Everyone has heard the story of the computer that tried to translate "The spirit is willing but the flesh is weak." into Russian and came out with something which meant "The vodka is strong but the meat is rotten." Unfortunately the problem is much more serious than just choosing the wrong words when translating idioms. It isn't always possible to even choose the right grammatical forms. We may want to translate the two sentences "A message was delivered by the next visitor." and "A message was delivered by the next day." If we are translating into a language which doesn't have the equivalent of our "passive voice", we may need to completely rearrange the first sentence into something corresponding to "The next visitor delivered a message." The other sentence might become something like "Before the next day, someone delivered a message." If the computer

picks the wrong form for either sentence, the meaning is totally garbled. In order to make the choice, it has to know that visitors are people who can deliver messages, while days are units of time and cannot. It has to "understand" the meaning of the words "day" and "visitor".

In other cases the problem is even worse. Even a knowledge of the meanings of words is not enough. Let us try to translate the two sentences:

"The city councilmen refused to give the women a permit for a demonstration because they feared violence." and

"The city councilmen refused to give the women a permit for a demonstration because they advocated revolution."

If we are translating into a language (like French) which has different forms of the word "they" for masculine and feminine, we cannot leave the reader to figure out who "they" refers to. The computer must make a choice and if it chooses wrong, the meaning of the sentence is changed. To make the decision, it has to have more than the meanings of words. It has to have the information and reasoning power to realize that city councilmen are usually staunch advocates of law and order, but are hardly likely to be revolutionaries.

For some uses, it isn't really necessary to understand much. A well known "psychiatrist" program named ELIZA is like this. It imitates the kind of Rogerian psychiatrist who would respond to a question like "What time is it?" by asking "Why do you want to know what

time it is?" or muttering "You want to know what time it is!". This doesn't take much understanding. All we need to do is take the words of the question and rearrange them in some simple way to make a new question or statement. In addition we might recognize a few key words, so we can respond with a fixed phrase whenever the patient uses one of them. If the patient types a sentence containing the word "mother", we can say "Tell me more about your family!". In fact, this is just how the psychiatrist program works. But very often it doesn't work -- its answers are silly or meaningless because it isn't really understanding what is being said.

If we really want computers to understand us, we need to give them much more intelligence. In addition to a grammar of the language, they need to have all sorts of knowledge about the subject they are discussing, and they have to use reasoning to combine facts in the right way to understand a sentence and respond to it. The process of understanding a sentence has to combine grammar, semantics, and reasoning in a very intimate way, calling on each part to help with the others.

This thesis explores one way of giving the computer knowledge in a flexible and usable form. In addition to a basic system for understanding language, we need to give the computer specialized information about the English language, the words we will use, and the subject we will discuss. In most earlier computer programs for understanding language, there have been attempts to use these kinds of

information in the form of lists of rules, patterns, and formulas. In our system, we express it as programs in special languages designed for syntax, semantics, and reasoning. This makes it possible to relate the different areas of knowledge more directly and completely. By using languages specially developed for representing these kinds of knowledge, it is possible for a person to "teach" the computer what it needs to know about a new subject or a new vocabulary without understanding the details of how the computer will go about using the knowledge to understand language. For simple information, it is even possible to just "tell" the computer in English. Other systems make it possible to "tell" the computer new things by limiting us to giving it very specialized kinds of information. By representing information as programs, we can greatly expand the range of things we can include.

The best way to experiment with such ideas is to write a working program using them, which can actually understand language. We would like a program which can answer questions, carry out commands, and accept new information in English. If we really want it to understand language, we must give it knowledge about the specific subject we want to talk about. For our experiment, we pretended that we were talking to a simple robot, with a hand and an eye and the ability to manipulate toy blocks on a table.

We can say "Pick up a block which is bigger than the one you are holding and put it in the box.", or ask a sequence of questions like "Had you touched any pyramid before you put the green one on the little

cube?" "When did you pick it up?" "Why?", or we can give it new information like "I like blocks which are not red, but I don't like anything which supports a pyramid." The "robot" responds by carrying out the commands (in a simulated scene on a display screen), typing out answers to the questions, and accepting the information to use in reasoning later on. (There are a number of hard technical problems in getting a computer to communicate by voice, and we have not attempted this. The dialog is carried out by typing on a terminal attached to the computer time-sharing system.)

We had three main kinds of goals in writing such a program. The first is the practical goal of having a language-understanding system. Even though we used the robot as our test area, the language programs do not depend on any special subject matter, and they have been adapted to other uses.

The second goal is gaining a better understanding of what language is and how it is put together. To write a program we need to make all of our knowledge about language very explicit, and we have to be concerned with the entire language process, not just one area such as syntax. We need the most advanced theories which linguists have developed, and we must fit them together to get the program working. This provides a rigid test for linguistic theories, and leads us into making new theories to fill the places where the old ones are lacking. To make a computer understand language, we have to understand better how people do it, and one important goal is to increase what we know about

language and linguistics.

More generally, we want to understand what intelligence is and how it can be put into computers. Language is one of the most complex and unique of human activities, and understanding its structure may lead to a better theory of how our minds work. The techniques needed to write a language-understanding program may be useful in many other areas of intelligence such as vision, mathematical problem solving, and game playing. All of these areas have similar problems of integrating large amounts of knowledge into a flexible system.. With advances in Artificial Intelligence we will some day be able to combine all of these areas to provide computers which are not tyrants, but can understand what we tell them and do what we want them to in an intelligent way.

1.1 General Description

This paper describes a system for the computer understanding of English. The system answers questions, executes commands, and accepts information in normal English dialog. It uses semantic information and context to understand discourse and to disambiguate sentences both syntactically and semantically. It combines a complete syntactic analysis of each sentence with a "heuristic understander" which uses different kinds of information about a sentence, other parts of the discourse, and general information about the world in deciding what a sentence means.

It is based on the belief that a computer cannot deal reasonably with language unless it can "understand" the subject it is discussing. The program is given a detailed model of a small part of knowledge, and it can understand sentences concerning this knowledge. We use as an example the knowledge needed by a simple robot with an eye and a hand. We can give it instructions to manipulate toy objects, interrogate it about the scene, and give it information it will use in deduction.

In addition to knowing the properties of blocks, hands, etc., the program has a simple model of its own mentality. It can remember and discuss its plans and actions as well as carry them out. It enters into a true dialog with a person, responding to English sentences with actions and English replies, and asking for clarification when its heuristic programs cannot disambiguate a sentence through use of context or physical knowledge.

1.1.1 What is Language?

To write a computer program which understands natural language, we need to understand what language is and what it does. We want to approach it not as a set of mathematical rules and symbols, but as a system intended to communicate ideas from a speaker to a hearer, and we want to analyze how it achieves that communication. It can be viewed as a process of translation from a structure of "concepts" in the mind of the speaker, into a string of sounds or written marks, and back into concepts in the mind of the hearer.

In order to talk about concepts, we must understand the importance of mental models (see Minsky (37)). In the flood of data pouring into our brains every moment, people recognize regular and recurrent patterns. From these we set up a model of the world which serves as a framework in which to organize our thoughts. We abstract the presence of particular objects, having properties, and entering into events and relationships. Our thinking is a process of manipulating the "concepts" which make up this model. In chapter 3 we show what this model might look like for a small area of knowledge, and describe how it can be used for reasoning.

When we communicate with others, we select concepts and patterns from the model and map them onto patterns of sound, which are then reinterpreted by the hearer in terms of his own model.

A theory can concentrate on either half of this process of generation and interpretation of language. Even though any theory must

be able to handle both, its approach is strongly colored by which one it views as logically primary. Most current theories are "generative", but it seems more interesting to look at the interpretive side (see Winograd (50) for a discussion of the issues involved). The first task a child faces is understanding rather than producing language, and he understands many utterances before he can speak any. At every stage of development, a person can understand a much wider range of patterns than he produces (see Miller, Chapter 7 (35)). This does not mean that in writing a program we are developing a detailed psychological theory of how a person interprets language, but there may in fact be very informative parallels.

Language understanding is a kind of intellectual activity, in which a pattern of sounds or written marks is interpreted into a structure of concepts in the mind of the interpreter. We cannot think of it as being done in simple steps: 1. Parse; 2. Understand the meaning; 3. Think about the meaning. The way we parse a sentence is controlled by a continuing semantic interpretation which guides us in a "meaningful" direction.

When we see the sentence "he gave the boy plants to water." we don't get tangled up in an interpretation which would be parallel to "He gave the house plants to charity." The phrase "boy plants" doesn't make sense like "house plants" or "boy scouts", so we reject any parsing which would use it.

Syntax, semantics, and inference must be integrated in a close way,

so that they can share in the responsibility for interpretation. Our program must incorporate the kind of flexibility needed for this kind of "vertical" system in which each part is constantly talking to the others. We have explored several techniques for integrating the large bodies of complex knowledge needed to understand knowledge. Two are particularly important.

First, we use a type of syntactic analysis which is designed to deal with questions of semantics. Rather than concentrating on the exact form of rules for shuffling around linguistic symbols, it studies the way language is structured around choices for conveying meaning. The parsing of a sentence indicates its detailed structure, but more important it abstracts the "features" of the linguistic components which are important for interpreting their meaning. The other parts of the program can look directly at these relevant features, rather than having to deal with minor details of the way the parsing tree looks.

Second, we represent knowledge in the form of procedures rather than tables of rules or lists of patterns. By developing special procedural languages for grammar, semantics, and deductive logic, we gain the flexibility and power of programs while retaining the regularity and understandability of simpler rule forms. Since each piece of knowledge can be a procedure, it can call on any other piece of knowledge of any type. The parser can call semantic routines to see whether the line of parsing it is following makes any sense, and the semantic routines can call deductive programs to see whether a

particular phrase makes sense in the current context. This is particularly important in handling discourse, where the interpretation of a sentence may depend in complex ways on the preceding discourse and knowledge of the subject matter.

This dual view of programs as data and data as programs would not have been possible in traditional programming languages. The ideas in the LISP language have been vital to making these ways of representing knowledge possible. The special languages for expressing facts about grammar, semantics, and deduction are embedded in LISP, and share with it the capability of ignoring the artificial distinction between programs and data.

1.1.2 Organization of the Language Understanding Process

We can divide the process of language understanding into three main areas -- syntax, semantics, and inference. As mentioned above, these areas cannot be viewed separately but must be understood as part of an integrated system. Nevertheless, we have organized our programs along these basic lines.

A. Syntax

First we need a system for the grammatical analysis of input sentences, or for kinds of phrases and other non-sentences we might want in our dialogs.

There have been many different parsing systems developed by different language projects, each based on a particular theory of grammar. The type of grammar chosen plays a decisive role in the type of semantic analysis which can be carried out. A language named PROGRAMMAR was designed specifically to fit the type of analysis used in this system. It differs from other parsers in that the grammar itself is written in the form of a collection of programs, and the parsing system is in effect an interpreter for the language used in writing those programs.

Having chosen the "type of grammar", we need to formalize a grammar for parsing sentences in a particular language. Our system includes a comprehensive grammar of English following the lines of systemic grammar (see Section 2.3). This type of grammar is well suited to a complete language-understanding system since it views language as a system for conveying meaning and is highly oriented toward semantic analysis. It

is intended to cover a wide range of syntactic constructions; the basic criterion for the completeness of the grammar is that a person with no knowledge of the system or its grammar should be able to type any reasonable sentence within the limitations of the vocabulary and expect it to be understood.

B. Inference

At the other end of the linguistic process we need a deductive system which can be used not only for such things as resolving ambiguities and answering questions, but also to allow the parser to use deduction in trying to parse a sentence. The system uses PLANNER, a deductive system designed by Carl Hewitt (see (23), (24), and (25)) which is based on a philosophy very similar to the general mood of this project. Deduction in PLANNER is not carried out in the traditional "logistic framework" in which a general procedure acts on a set of axioms or theorems expressed in a formal system of logic. Instead, each theorem is in the form of a program, and the deductive process can be directed to any desired extent by "intelligent theorems." PLANNER is actually a language for the writing of those theorems.

This deductive system must be given a model of the world, with the concepts and knowledge needed to make its deductions. Useful language-understanding can occur only when a program (or person) has an adequate understanding of the subject he is talking about. We will not attempt to understand arbitrary sentences talking about unknown subjects, but instead will give the system detailed knowledge about a particular

subject -- in this case, the simple robot world of children's toy blocks and some other common objects. The deductive system has a double task of solving goal-problems for a robot within this world, and then talking about what it is doing and what the scene looks like. We want the robot to discuss its plans and actions as well as carry them out. We can ask questions not only about physical happenings, but about the robot's goals as well. We can ask "Why did you clear off that block?" or "How did you do it?". This means that the model includes not only the properties of blocks, hands, and tables, but a model of the robot mind as well. We have written a collection of PLANNER theorems and data called BLOCKS, describing the world seen by the robot, and the knowledge it needs to work with that world. (see Section 3.4). See Figure 1 for a picture of a typical scene.

C. Semantics

To connect the syntactic form of the sentence to its meaning, we need a semantic system. This again calls for a general system, and a language in which we can easily express the meanings of words and grammatical constructions. The system includes mechanisms for setting up simple types of semantic networks and using deductions from them as a first phase of semantic analysis. For example, the network could include the information that a "block" is a physical object, while a "bloc" is a political object, and the definition of the word "support" could use this information in choosing the correct meanings for the sentences

THE ROBOT'S WORLD

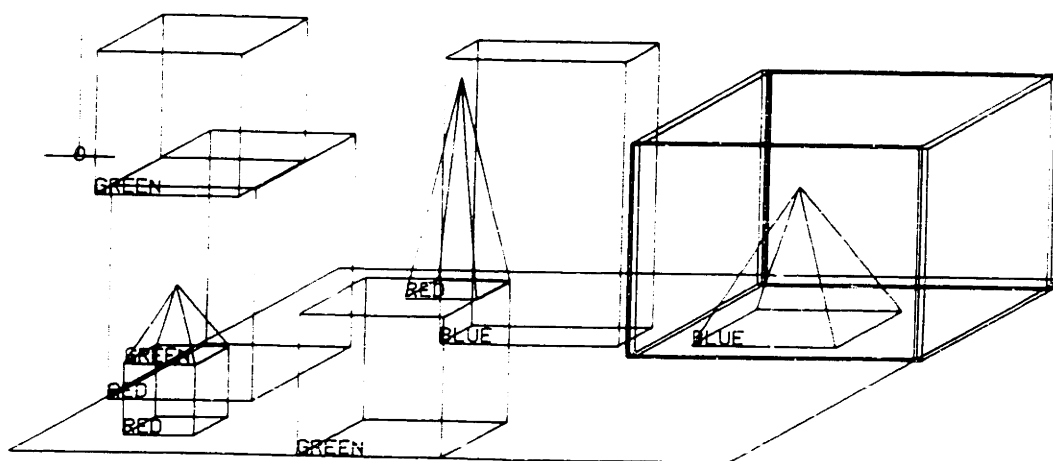


Figure 1

The red block supports the pyramid. and

The red bloc supports Egypt.

More important, the meaning of a word or construction is also defined in the form of a program to be interpreted in a semantic language. It is this aspect of semantics which is missing in most other theories, which limit themselves to a particular type of network or relational structure. The meaning selected for a word can depend on any aspect of the sentence, the discourse, or the world. In deciding on the meaning of "one" in "Pick up the green one." we need a program which can examine past sentences. This program is actually included as part of the definition of the word "one".

Part of this semantic system must include a powerful heuristic program for resolving ambiguities and determining the meaning of references in discourse. In almost every sentence, reference is made either explicitly (as with pronouns) or implicitly (as with the word "too") to objects and concepts not explicitly mentioned in that sentence. To interpret these, the program must have at its disposal not only a detailed grammatical analysis (to check for such things as parallel constructions), but also a powerful deductive capacity (to see which reference assignments are logically plausible), and a thorough knowledge of the subject it is discussing (to see which interpretations are reasonable in the current situation).

In order to deal with language in a human way, we must take into account all sorts of discourse knowledge. In addition to remembering

the immediately previous sentences for such things as pronoun references, the system must remember what things have been mentioned throughout the discussion, so that a reference to "the pyramid" will mean "the pyramid we mentioned earlier" even if there are several pyramids in the scene.

In addition the system must have some model of the knowledge of the person communicating with it. If we say "Is there a block on a green table?" "What color is it?", the word "it" refers to the block. But if we had asked "Is there a green block on a table?" "What color is it?", "it" must refer to the table since we would not ask a question which we had answered ourselves in the previous sentence.

Our semantic system works with a base of knowledge about simple semantic features in the subject domain, and with a collection of definitions for individual words. These definitions are written in a "semantics language" which allows simple words to be defined in a straightforward way, while allowing more complex words to call on arbitrary amounts of computation to integrate their meaning into the sentence.

Finally we need a generative language capacity to produce answers to questions and to ask questions when necessary to resolve ambiguities. Grammatically this is much less demanding than the interpretive capacity, since humans can be expected to understand a wide range of responses, and it is possible to express almost anything in a syntactically simple way. However, it takes a sophisticated semantic

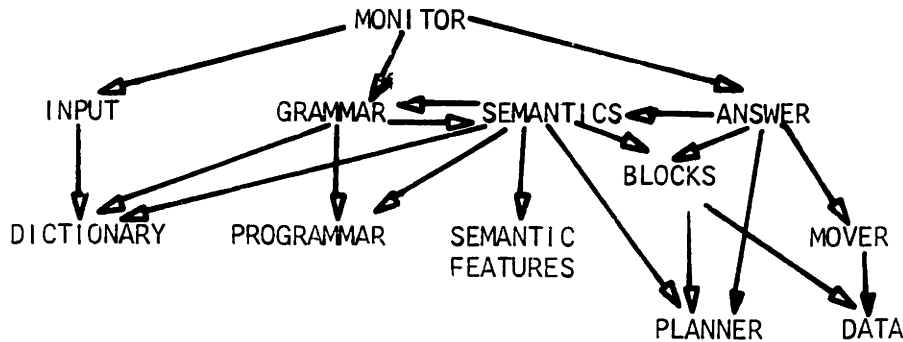
and deductive capability to phrase things in a way which is meaningful and natural in discourse.

Listing these aspects of language understanding separately is somewhat misleading, as it is the interconnection and interplay between them which makes the system possible. Our parser does not parse a sentence, then hand it off to an interpreter. As it finds each piece of the syntactic structure, it checks its semantic interpretation, first to see if it is plausible, then (if possible) to see if it is in accord with the system's knowledge of the world, both specific and general. This has been done in a limited way by other systems, but in our program it is an integral part of understanding at every level.

1.2 Implementation of the System

The language understanding program is written in LISP to run under the PDP-10 Incompatible Time-sharing System at the Project MAC Artificial Intelligence Group at MIT. When operating with a 200 word vocabulary and a fairly complex scene, it occupies approximately 80K of core. This includes the LISP interpreter, all of the programs, dictionary entries, and data, and enough free storage to remember a sequence of actions and to handle complex sentences and deductions.

The program is organized as follows (Arrows indicate that one part of the program calls another directly):



1. MONITOR is a small LISP program which calls the basic parts of the system. Since the system is organized vertically, most of the communication between components is done directly, and the monitor is called only at the beginning and end of the understanding process.

2. INPUT is a LISP program which accepts typed input in normal English orthography and punctuation, looks up words in the dictionary, performs morphemic analysis (e.g. realizing that "running" is the "ing"

form of the word "run", and modifying the dictionary definition accordingly), and returns a string of words, together with their definitions. This is the input with which the grammar works.

3. The GRAMMAR is the main coordinator of the language understanding process. It consists of a few large programs written in PROGRAMMAR to handle the basic units of the English language (such as clauses, noun groups, prepositional groups, etc.).

The system has two PROGRAMMAR compilers, one which compiles into LISP, which is run interpretively for easy debugging, and another which makes use of the LISP compiler to produce LAP assembly code for efficiency.

4. SEMANTICS is a collection of LISP programs which work in coordination with the GRAMMAR to interpret sentences. In general there are a few semantics programs corresponding to each basic unit in the grammar, each performing one phase of the analysis for that unit. These semantics programs call PLANNER to make use of deduction in interpreting sentences.

5. ANSWER is another collection of LISP programs which control the responses of the system, and take care of remembering the discourse for future reference. It contains a number of heuristic programs for producing answers which take the discourse into account, both in deciding on an answer and in figuring out how to express it in fluent English.

6. PROGRAMMAR is a parsing system which interprets grammars written in the form of programs. It has mechanisms for building a parsing tree,

and a number of special functions for exploring and manipulating this tree in the GRAMMAR programs. It is written in LISP.

7. The DICTIONARY actually consists of two parts. The first is a set of syntactic features associated with each word, used by the GRAMMAR. The second is a semantic definition for each word, written in a language which is interpreted by the SEMANTICS programs. The form of a word's definition depends on its type (e.g. the definition of "two" is "2"). There are special facilities for definitions of irregular forms (like "geese" or "slept"), and only the definitions of root words are kept, since INPUT can analyze endings of all different types. The definitions are actually kept on the LISP property list of the word, and dictionary lookup is handled automatically by LISP.

8. The system has a network of SEMANTIC FEATURES, kept on property lists and used for an initial phase of semantic analysis. The features subdivide the world of objects and actions into simple categories, and the semantic interpreter uses these categories to make some of its choices between alternative definitions for a word (for example "in" would mean "contained in" when applied to a hollow object like a box, but would mean "part of" when applied to a conglomerate like a stack, as in "the red block in the tall stack").

9. BLOCKS is a collection of PLANNER theorems which contain the system's knowledge about the properties of the physical world, and the methods of achieving goals and making deductions about it. It is a very simple model, designed less for a realistic simulation of a robot, and

more to give the system a world to talk about in which one can say many different types of things.

10. `MOVER` is a set of display routines written in `LAP` which simulate the robot's world on the `DEC 340` graphic display attached to the `PDP-10` computer. It displays the scene indicated by the `DATA` currently in memory, and any action called for is displayed in real time, so the human can get visual feedback from his commands to the robot. The input sentences and responses are also displayed. Figure 1 displays a typical scene.

11. `PLANNER` is the deductive system used by the program, and is used at all stages of the analysis, both to direct the parsing process and to deduce facts about the `BLOCKS` world. Currently the system uses `Micro-Planner`, an abbreviated version of Carl Hewitt's original language, implemented by Terry Winograd, Gerald Sussman, and Eugene Charniak (reference (51)).

It is written in `LISP`.

12. The `DATA` about the current scene is in the form of `PLANNER` assertions describing the objects, their sizes, shapes, colors, and locations. The system assumes it has some sort of robot "eye", so all of this information is initially known and does not have to be entered in English.

When running compiled, the system is fast enough to carry on a quite reasonable real-time discourse. Each sentence takes from 5 to 20 seconds to analyze and respond to, and the display is designed to move

at the speed of a real arm.

The program listings are all available in hard copy and DECtape, but before implementing them on another system, it would be wise to read about the special features of MACLISP (Reference (58))

which may be in conflict with other LISP implementations.

1.3 Sample Dialog

This section contains a sample of a dialog with the language understanding system. It carries out actions and answers questions about a simple scene containing a table, hand, and box, and several blocks and pyramids. The figures in this chapter show the actions taken during the dialog.

The lines in lower case were typed in by a person, and the lines in upper case are the computer's response. Lines beginning with a semicolon are comments describing some of the features being exhibited.

The dialog was carried out in real time with the response of the "robot" displayed on a CRT screen. Some figures are included to show the state of the scene at various points.

1. pick up a big red block.

OK.

```
;The system answers "OK" when it carries out a
;command. In order to pick up the red block, it had
;to clear it off by finding a space for the green
;one and moving it away.
;Figure 2 shows the scene when the command was given,
;Figure 3 shows the green block being put out of the
;way, and Figure 4 shows the end result.
```

2. grasp the pyramid.

I DON'T UNDERSTAND WHICH PYRAMID
YOU MEAN.

```
;There are three pyramids in the scene.
```

3. find a block which is taller than the one
you are holding and put it into the box.

```
;The system has to figure out what is meant by "one"
;and "it", as well as using deduction to decide which
```

PICK UP A BIG RED BLOCK.

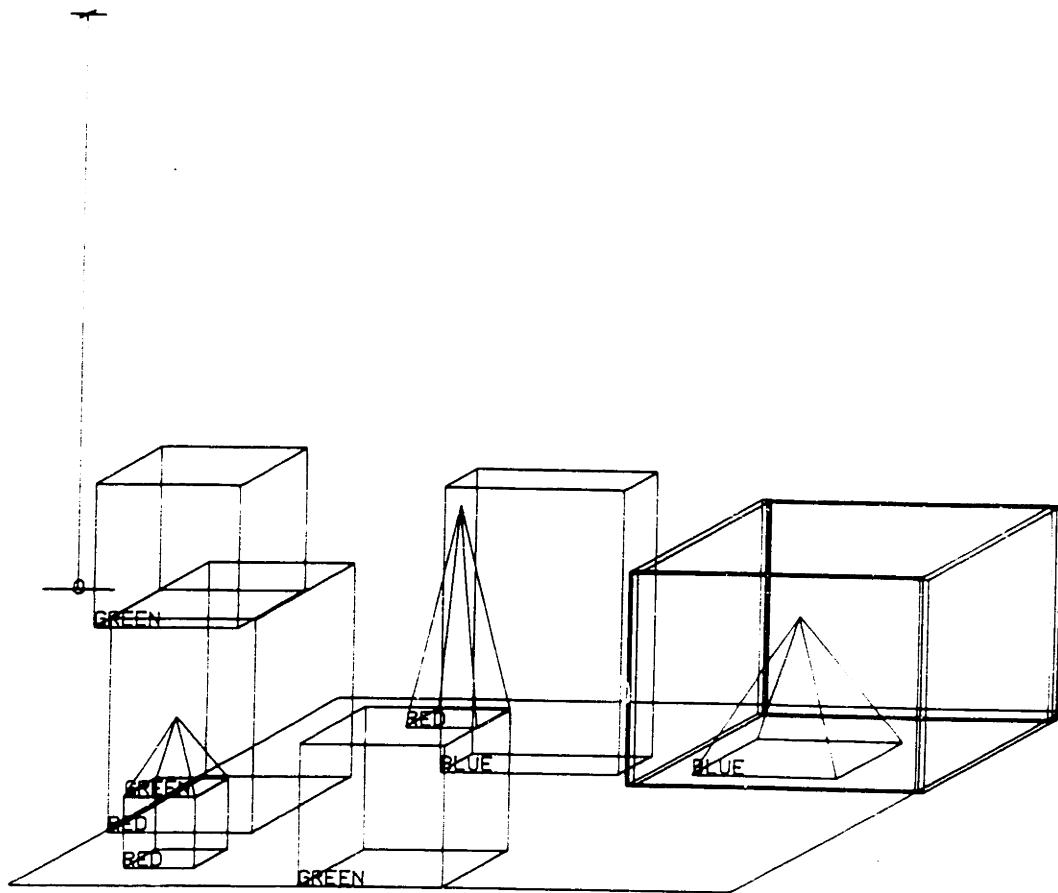


Figure 2

PICK UP A BIG RED BLOCK.

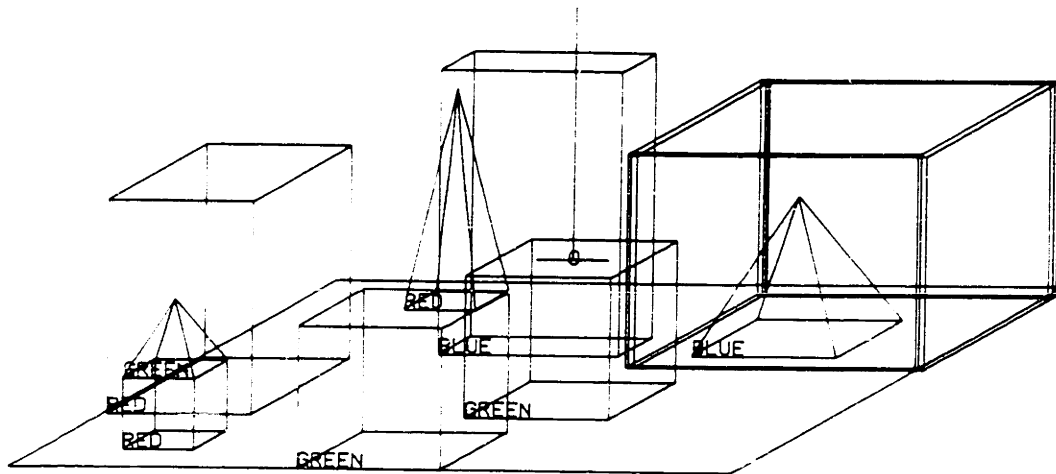


Figure 3

PICK UP A BIG RED BLOCK.

OK.

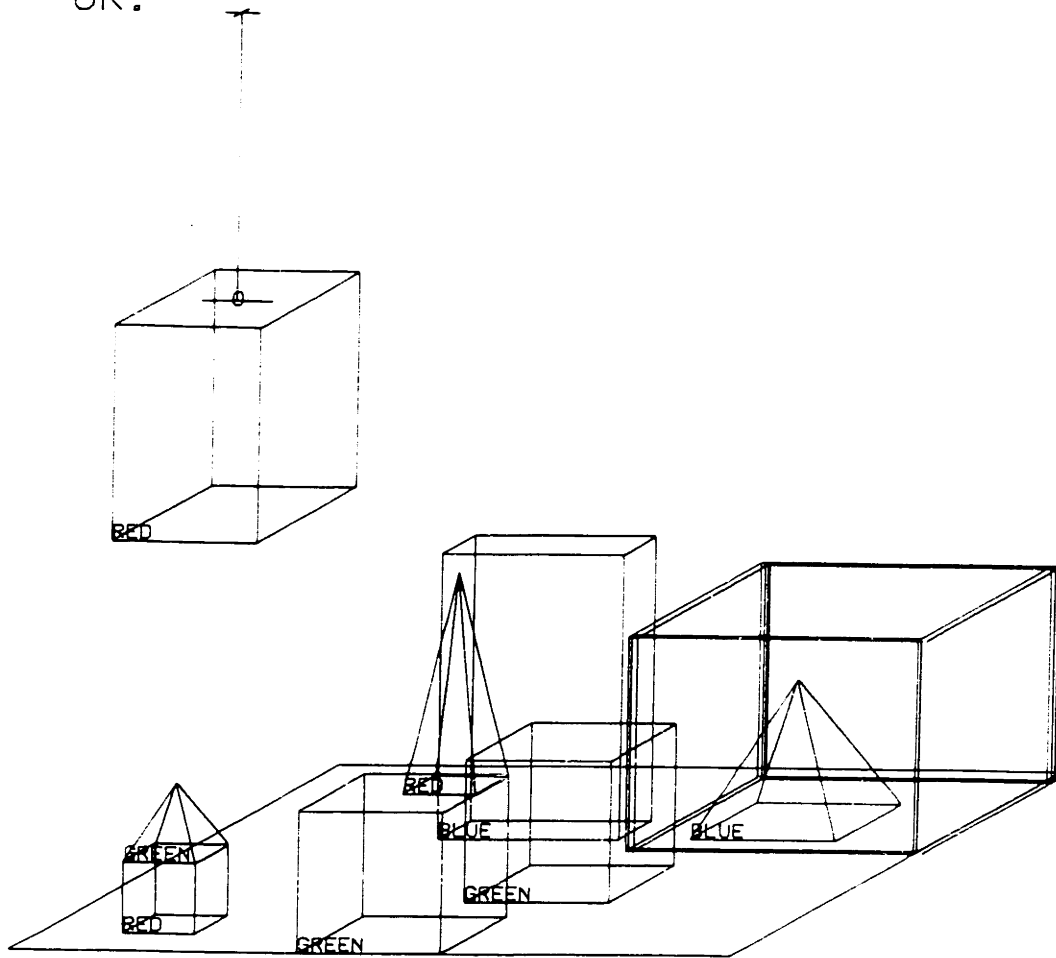


Figure 4

;block it is holding.

BY "IT", I ASSUME YOU MEAN THE BLOCK
WHICH IS TALLER THAN THE ONE I AM
HOLDING.

;Printed out to keep us informed.

OK.

;The hand has to set down the red block to pick up
;the taller blue one. Figure 5 shows the result.

4. what does the box contain?

THE BLUE PYRAMID AND THE BLUE BLOCK.

5. what is the pyramid supported by?

;Earlier the system responded to "the pyramid" by saying
;"I don't understand which pyramid you mean." now,
; it knows that we must mean the one which was just
;mentioned.

THE BOX.

6. how many blocks are not in the box?

;Negatives such as "not" and "no" are handled whether
;they appear in clauses or noun groups (such as "no boxes").

FOUR OF THEM.

7. is at least one of them narrower than
the one which i told you to pick up?

;The phrase "at least one of them" must
;be analyzed to refer back to the previous question, and
;"the one which..." must also refer to blocks. Noun
;groups can have complicated numbers such as "at
;least one", "more than three", "exactly seven", etc.
;The memory of past events is used to deduce which
;block is "the one i told you to pick up". Notice
;the complicated grammatical structure in which "the one"
;is actually the object of a clause embedded two levels
;deeper.

YES, THE RED CUBE.

8. is it supported?

;Passives can be used without a "by ..." phrase.

FIND A BLOCK WHICH IS TALLER THAN THE ONE
YOU ARE HOLDING AND PUT IT INTO THE
BOX.
BY "IT", I ASSUME YOU MEAN THE BLOCK
WHICH IS TALLER THAN THE ONE I AM
HOLDING.
OK.

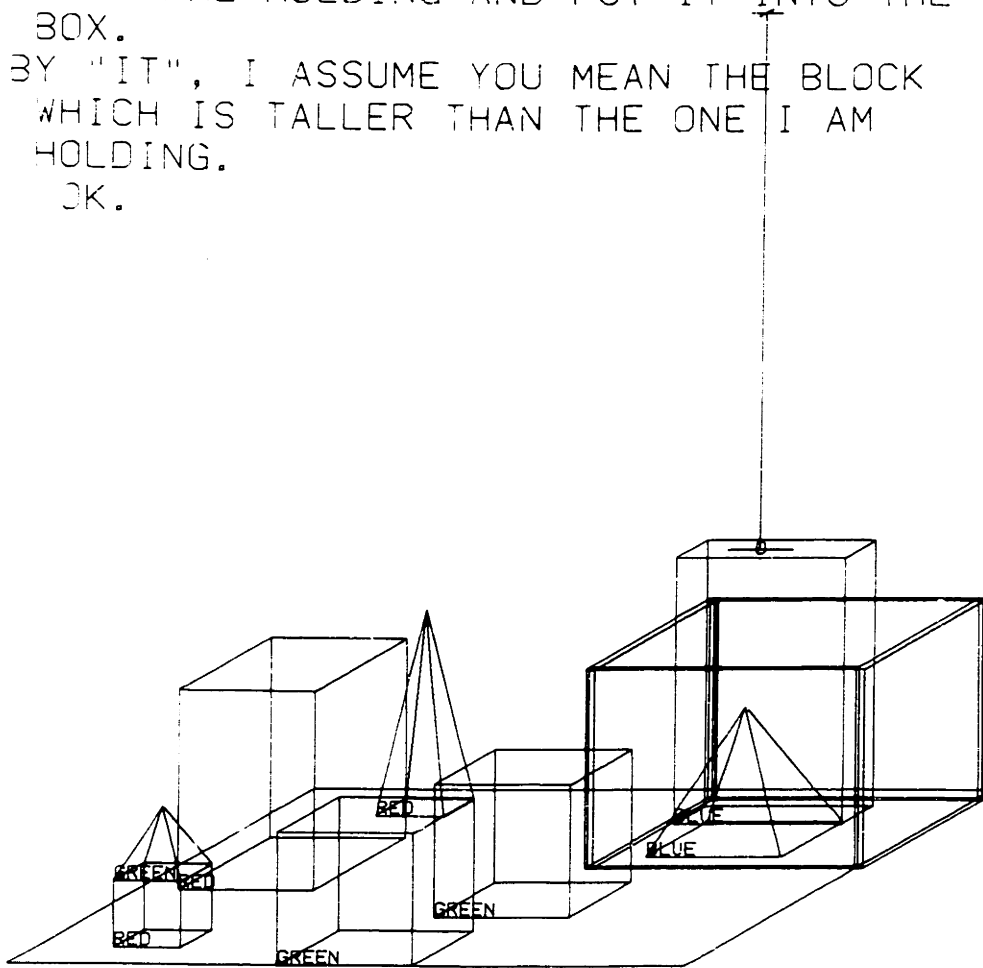


Figure 5

YES, BY THE TABLE.

9. can the table pick up blocks?

NO.

;The semantic rules make this a nonsensical question
;which therefore must be false.

10. can a pyramid be supported by a block?

YES.

;The deductive system finds an actual example, so
;it knows this is possible.

11. can a pyramid support a pyramid?

I DON'T KNOW.

;The question makes sense, but there is no example
;in the scene, so there is no easy way to tell without
;being able to examine the robot's programs.

12. stack up two pyramids.

I CAN'T.

;The robot tries and fails.

13. the blue pyramid is nice.

I UNDERSTAND.

;This data is added to the data base.

14. i like blocks which are not red,
but i don't like anything which
supports a pyramid.

I UNDERSTAND.

;Two new theorems are created for proving things about "liking".

15. do i like the box?

NO.

;The box supports whatever it contains, including
;a pyramid.

16. do i like anything in the box?

YES, THE BLUE BLOCK AND THE
BLUE PYRAMID.

;Presumably we like anything which is nice, and the
;blue block is certainly not red.

17. will you please stack up both of the
red blocks and either a green cube or
a pyramid?

OK.

;Logical connectives such as "and" "or" "either"
;etc. are handled in both the grammar and semantics.
;A command is executed even if it looks like a question.
;The robot carries out the command by the series of steps
;shown in figures 6 through 9.
;It puts a green cube on the large red block
;(note that it chooses the green cube with nothing
;on it), then removes the small pyramid from the
;little red cube, so that it can use that cube
;to complete the stack as specified.

18. which cube is sitting on the table?

THE LARGE GREEN ONE WHICH SUPPORTS
THE RED PYRAMID.

;Ordinarily the system would be satisfied with
;calling this "a big green cube", but since we asked
;"which cube" it has to be more specific in naming it.
;The use of substitute nouns (such as "one") and pronouns
;in answers makes the discourse more natural.

19. is there a large block behind a pyramid?

YES, TWO OF THEM, A LARGE RED ONE
AND THE BLUE ONE.

20. put a small one onto the green cube
which supports a pyramid.

;"A small one" must contrast with the earlier phrase
;"a large block", so this means "a small block".

OK.

21. put the littlest pyramid on top of it.

WILL YOU PLEASE STACK UP BOTH OF THE
RED BLOCKS AND EITHER A GREEN CUBE OR
A PYRAMID?

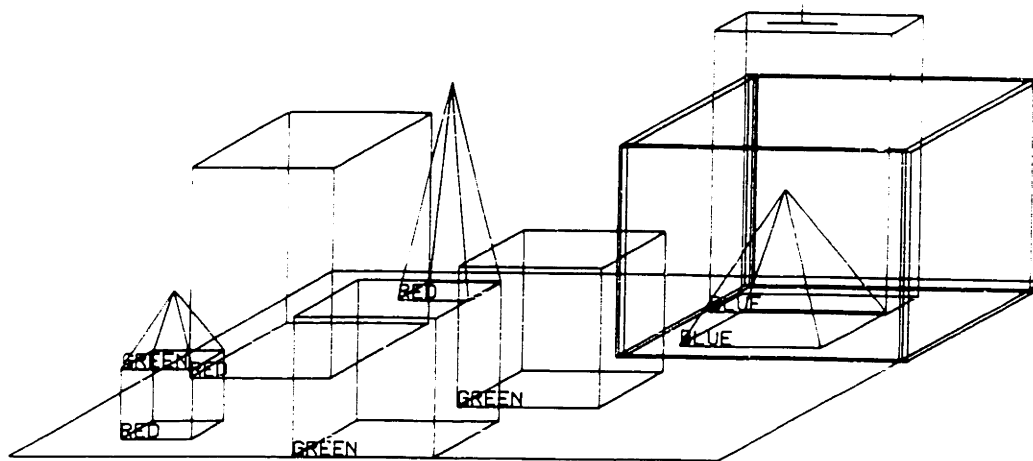


Figure 5

WILL YOU PLEASE STACK UP BOTH OF THE
RED BLOCKS AND EITHER A GREEN CUBE OR
A PYRAMID?

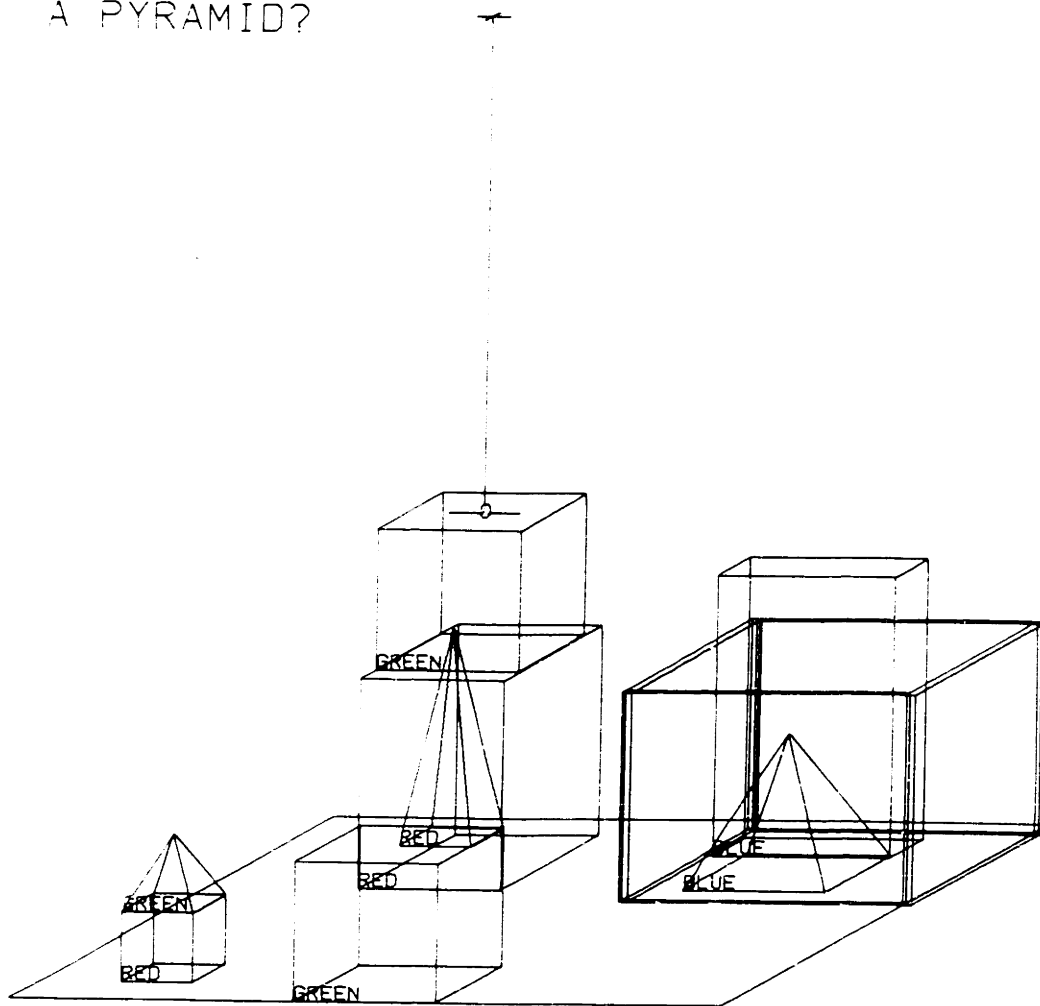


Figure 7

WILL YOU PLEASE STACK UP BOTH OF THE
RED BLOCKS AND EITHER A GREEN CUBE OR
A PYRAMID?

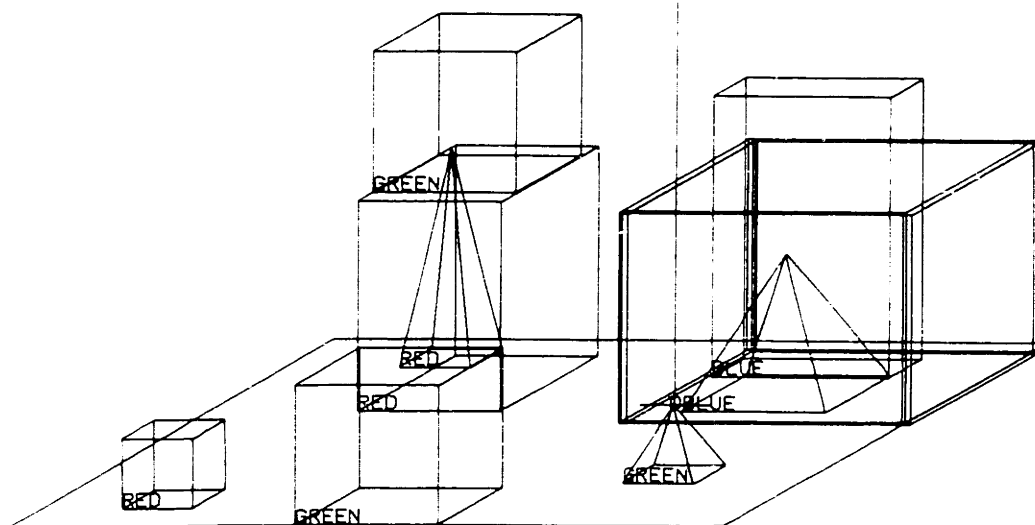


Figure 8

WILL YOU PLEASE STACK UP BOTH OF THE
RED BLOCKS AND EITHER A GREEN CUBE OR
A PYRAMID?

OK.

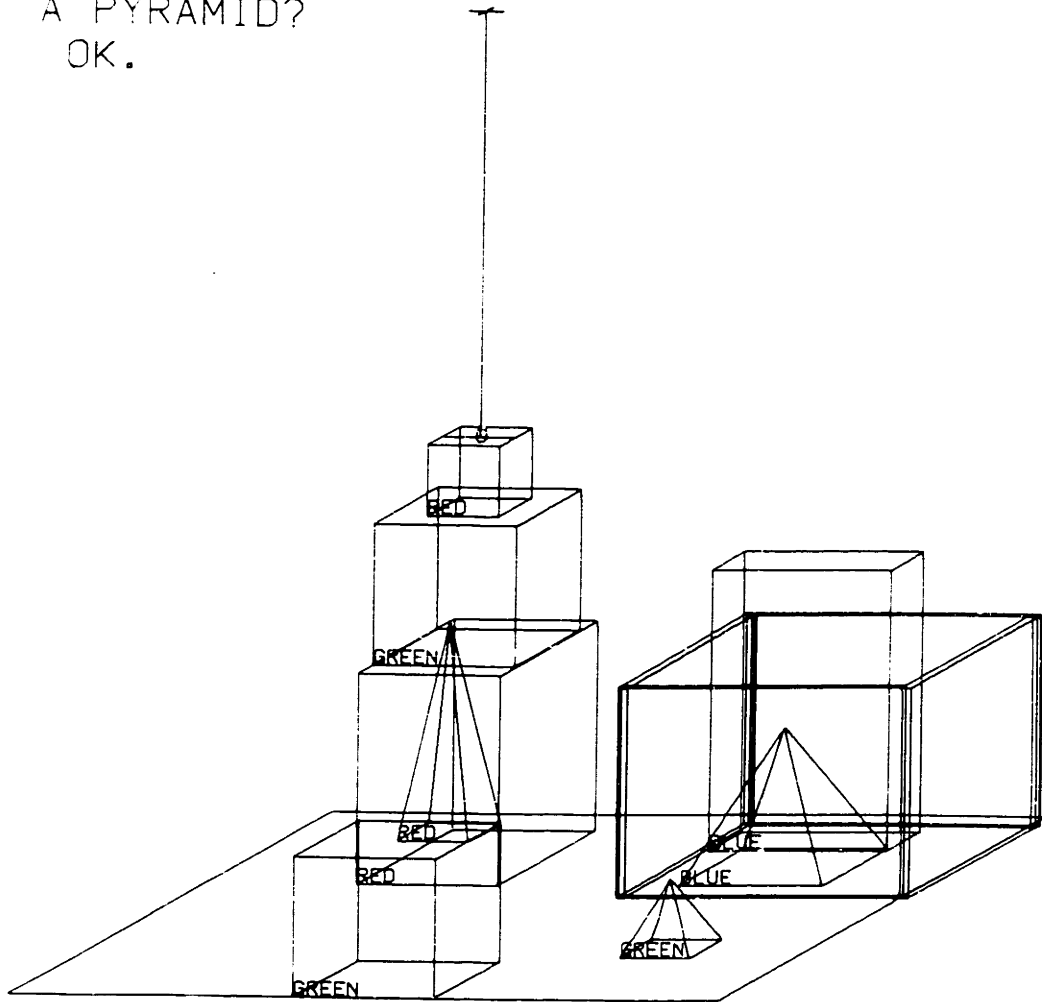


Figure 9

;Words like "littlest" are not in the dictionary but
 ;are interpreted from the root forms like "little".
 ;The scene now looks like Figure 10.

OK.

22. does the shortest thing the tallest
 pyramid's support supports support
 anything green?

;The grammar can handle words like "support" which are both
 ;verbs and nouns without getting confused, and can
 ;understand possessives. To compute the answer, see
 ;Figure 11 for the scene.

YES, THE GREEN PYRAMID.

23. what color is it?

BY "IT", I ASSUME YOU MEAN THE SHORTEST
 THING THE TALLEST PYRAMID'S SUPPORT
 SUPPORTS.

;If we had asked "what size is it?", the system would
 ;have assumed that "it" meant the green pyramid,
 ;but nobody would ask the color of something
 ;which was just called "the green pyramid", so it
 ;had to look farther for the reference of "it".

RED

24. how many things are on top of green
 cubes?

I'M NOT SURE WHAT YOU MEAN BY "ON TOP
 OF "IN THE PHRASE "ON TOP OF GREEN
 CUBES " .

DO YOU MEAN:

- 1 - DIRECTLY ON THE SURFACE
- 2 - ANWHERE ON TOP OF ?

;When there are several interpretations and noe is
 ;clearly better, the system has to ask the user for
 ;more information. The paraphrases were included with the
 ;dictionary definition of "on", and the request for more
 ;information occurs automatically when the heuristics for
 ;resolving ambiguities aren't suficient. A request like
 ;this can also occur when the meaning of a pronoun such as

PUT THE LITTLEST PYRAMID ON TOP OF IT.

OK.

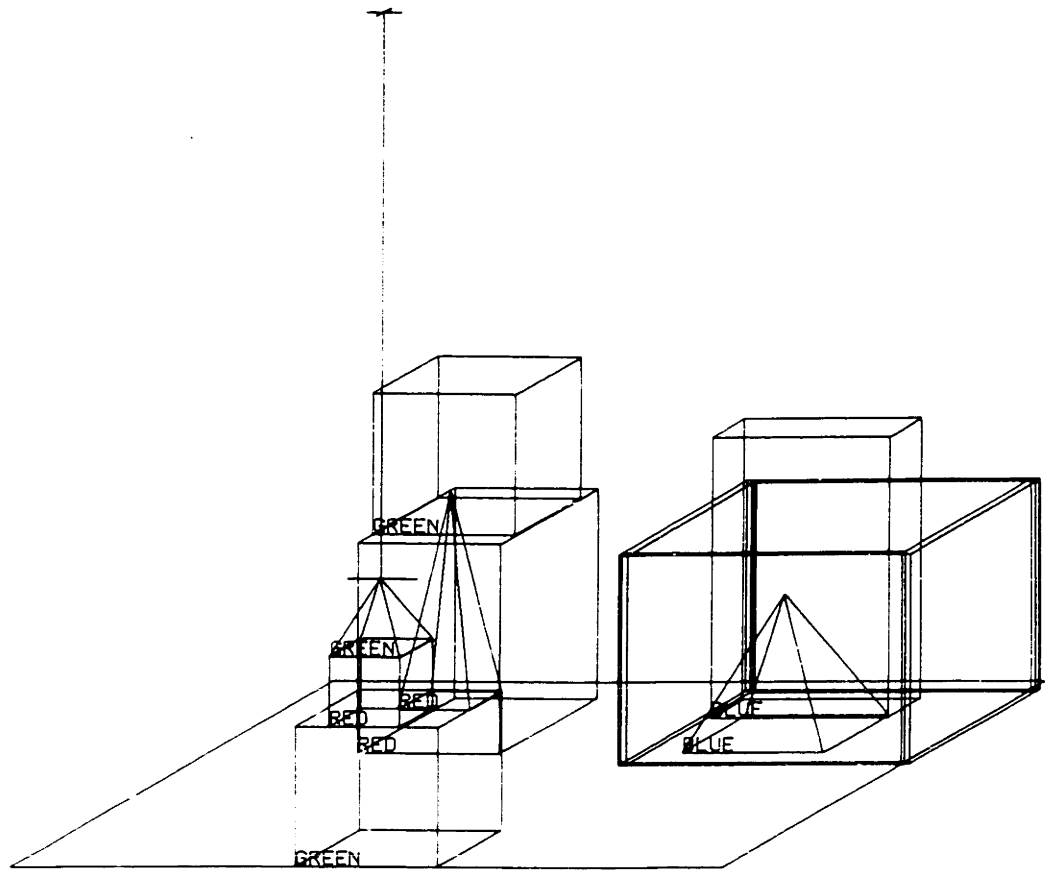


Figure 10

DOES THE SHORTEST THING THE TALLEST
PYRAMID'S SUPPORT SUPPORTS SUPPORT
ANYTHING GREEN?

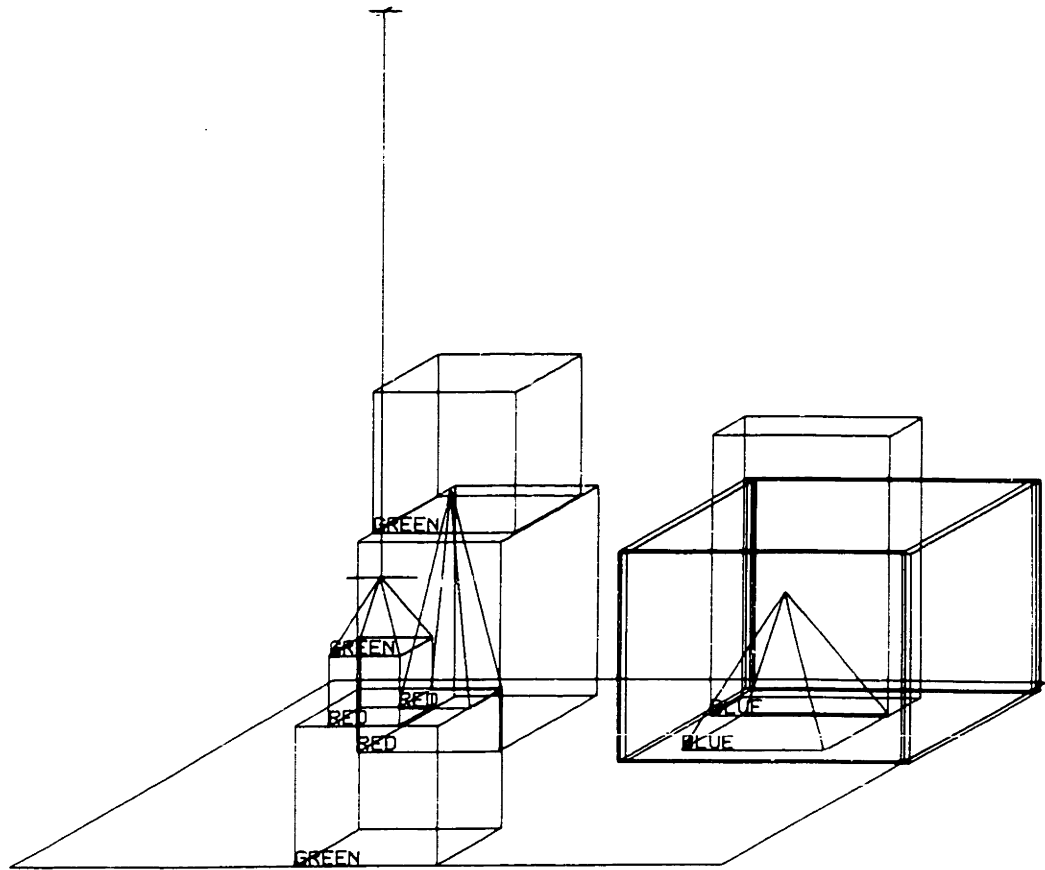


Figure 11

;"it" or "they" is unclear.

2

THREE OF THEM.

25. had you touched any pyramid before
you put the green one on the little
cube?

YES, THE GREEN ONE.

;The system deduces from its memory what event is being
;referred to, and can relate other events to it in time.
;The analysis includes complex tenses of verb groups, and
;there are special facilities for keeping track of the various
;forms of irregular verbs like "have"

26. when did you pick it up?

WHILE I WAS STACKING UP THE RED CUBE,
A LARGE RED BLOCK AND A LARGE GREEN
CUBE.

;A time is described by naming the main goal which was being
;worked on.

27. why?

;The system remembers immediately previous sentences
;to understand questions like this and the following ones.

TO GET RID OF IT.

;By keeping track of selected parts of the original
;subgoal tree, the system has some understanding of its
;own motives.

28. why did you do that?

TO CLEAN OFF THE RED CUBE.

;"That" refers to the action mentioned in the answer.

29. why did you clear off that cube?

;Here, "that" means the one just referred to.

TO PUT IT ON A LARGE GREEN CUBE.

30. why did you do that?

TO STACK UP THE RED CUBE , A LARGE
RED BLOCK AND A LARGE GREEN CUBE.

31. why did you do that?

BECAUSE YOU ASKED ME TO.

;It remembers what it did, not how the request was
;worded.

32. how did you do it?

;"It" refers to the action mentioned in the last
;question, rather than the answer.

BY PUTTING A LARGE RED BLOCK ON THE
TABLE ; THEN LETTING GO OF IT ; THEN
PUTTING A LARGE GREEN CUBE ON IT ;
THEN LETTING GO OF THAT CUBE ; THEN
PUTTING THE RED CUBE ON THAT CUBE ;
THEN LETTING GO OF THAT CUBE.

;This is a description of the top-level subgoals
;used to achieve the goal.

33. how many objects did you touch while
you were doing it?

;Time references can refer to a limit time span,
;using words like "while".

FOUR OF THEM.

34. what did the red cube support before
you started to clean it off?

THE GREEN PYRAMID.

;By remembering what has been moved, it is possible
;to deduce things about earlier states of the scene.
;We can use complex nestings of clauses like "before
;you started to clean it off".

35. there were five blocks to the left of
the box then.

;"Then" must refer to the time mentioned in the last
;sentence.

NO, ONLY FOUR OF THEM: THE RED CUBE ,
TWO LARGE GREEN CUBES AND A LARGE RED
BLOCK.

;Earlier in the dialog, information about "liking"
;was accepted at face value. But the system knows that
;it has complete information about about the basic properties

;of objects, including location. It checks any statements
 ;about location or other such properties to see whether it
 ;agrees or disagrees.

36. put the blue pyramid on the block in
 the box.

;This is syntactically ambiguous, but is understood without
 ;ambiguity since the parsing corresponding to "put the
 ;blue pyramid which is on the block in the box." does not
 ;make sense in the current scene, while "put the blue pyramid
 ;on the block which is in the box" does. We can see
 ;the situation in Figure 12, and the result in Figure 13.

OK.

37. is there anything which is bigger
 than every pyramid but is not as wide
 as the thing that supports it?

;The system understands quantifiers such as "every", "any", "some",
 ;etc., and structures such as "bigger than", "as wide as", etc.
 ;Here "it" refers to a noun group within the same sentence
 ;but two levels up in the parsing tree.

YES, THE BLUE BLOCK.

39. does a steeple
 SORRY, I DON'T KNOW THE WORD "STEEPLE".
 PLEASE TYPE TWO SPACES.

;The system interrupts immediately when it sees a word it
 ;can't figure out. The two spaces are to clear the input
 ;buffer of anything typed after the message was sent but
 ;before the typist saw it.

40. a "steeple" is a stack which contains
 two cubes and a pyramid.

;The quote marks indicate that we are defining a new word.

I UNDERSTAND.

41. are there any steeples now?

NO.

;The new definition is used as a description to recognize
 ;steeples.

42. build one.

PUT THE BLUE PYRAMID ON THE BLOCK IN THE BOX.

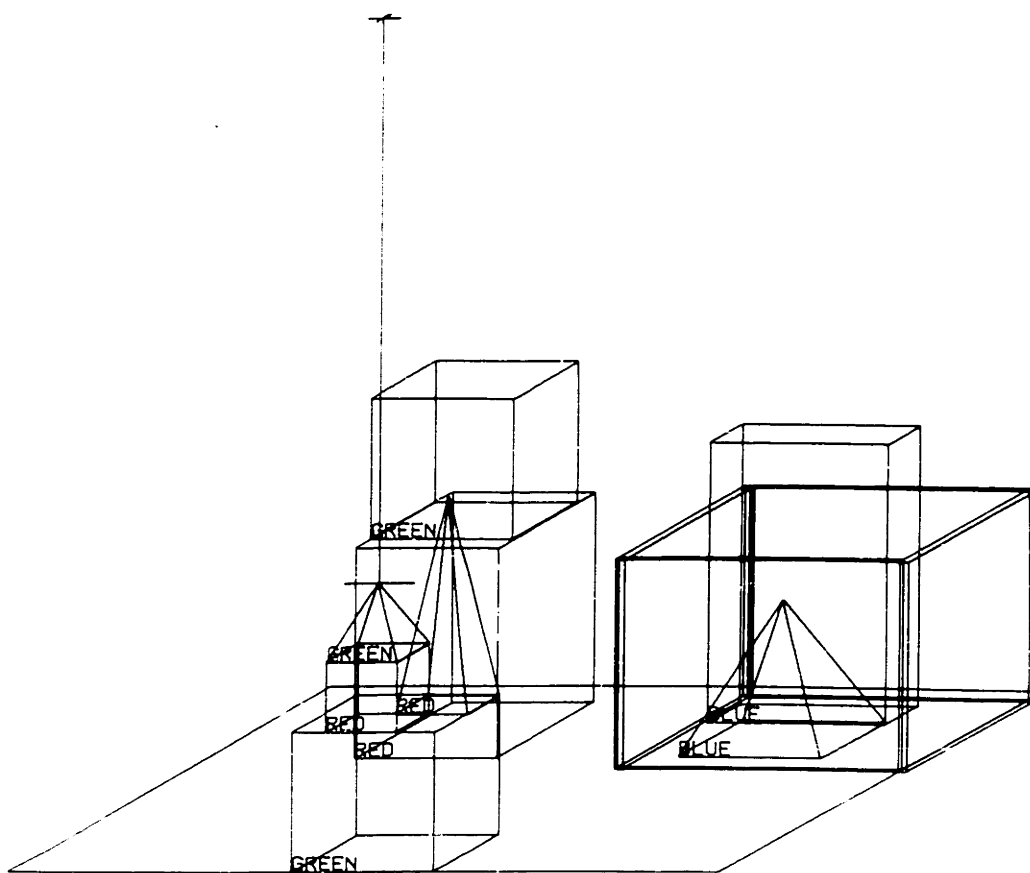


Figure 12

PUT THE BLUE PYRAMID ON THE BLOCK IN THE BOX.

OK.

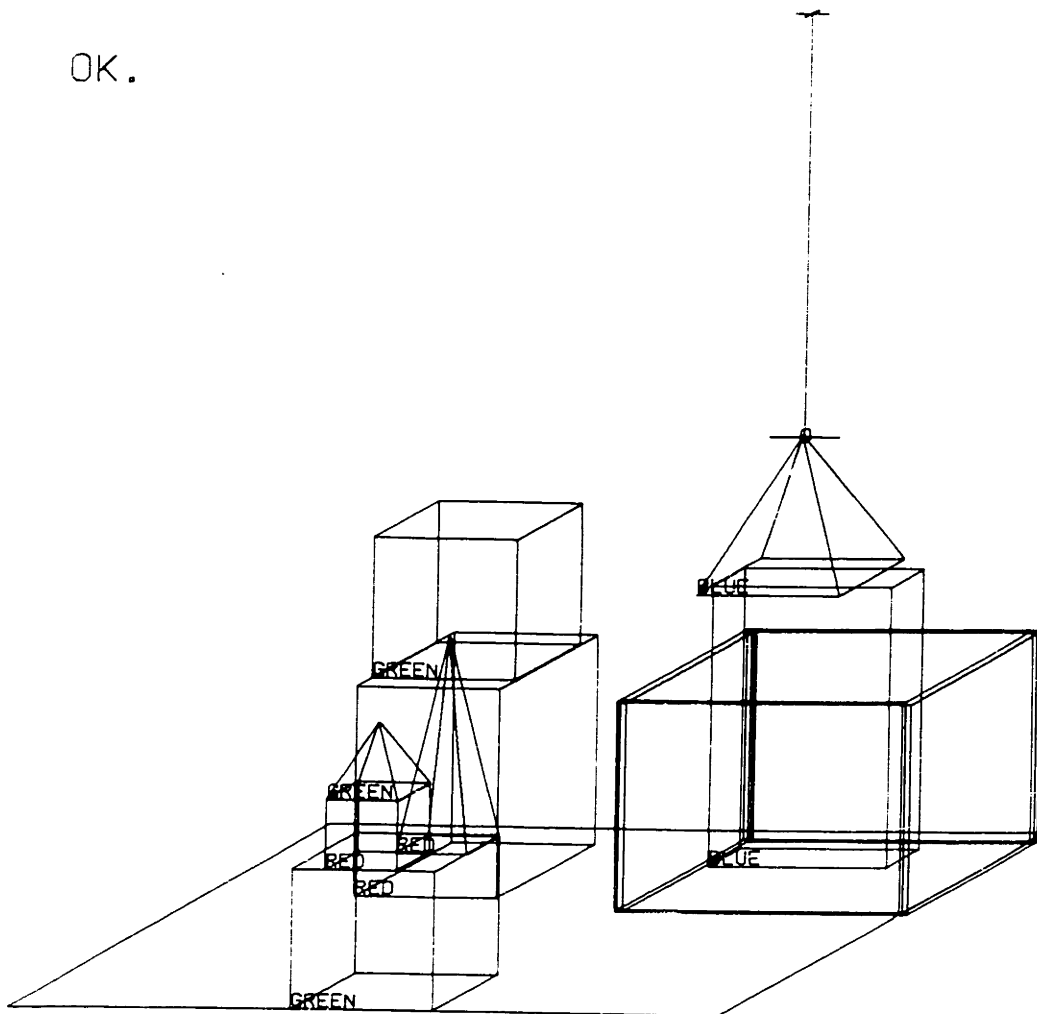


Figure 13

OK.

;The description given in the definition is enough to
 ;deduce a plan for building new objects as well as recognizing
 ;them. The word "one" means "a steeple", and the plan is
 ;carried out. First, one of the large green cubes is cleared
 ;off by removing the red cube and two pyramids.
 ;Then the two green cubes and the tall red pyramid
 ;are stacked up to fit the description.

43. call the biggest block "superblock".

OK.

;We can also give objects names.

44. have you picked up superblock since
 we began?

YES.

45. why did you drop it?

BY "IT", I ASSUME YOU MEAN THE
 SUPERBLOCK.

I CAN'T EXPLAIN A NON-EXISTENT EVENT.

;There is no record in memory of any block being dropped.

46. is there anything to the right of
 the red pyramid?

YES, FOUR THINGS: THE BOX , THE RED
 CUBE, THE BLUE PYRAMID AND THE BLOCK
 NAMED SUPERBLOCK

;Once an object has a name, the name is used to help
 ;identify it in responses.

47. thank you.

YOU'RE WELCOME!

;The special idiom-defining capabilities allow us to
 ;include such pleasantries without modifying the grammar.

2. Syntax

2.1 Basic Approach to Syntax

2.1.1 Syntax and Meaning

The decision to consider syntax as a proper study devoid of semantics is a basic tenet of most current linguistic theories. Language is viewed as a way of organizing strings of abstract symbols, and competence is explained in terms of symbol-manipulating rules. At one level this has been remarkably successful. Rules have been formulated which describe in great detail how most sentences are put together. The problem comes in trying to relate those symbols and structures to the function of language as a vehicle for conveying meaning. The same approach which has worked so well in accounting for the machinations of syntax has been unable to provide any but the most rudimentary and unsatisfactory accounts of semantics.

The problem is not that current theories are finding wrong answers to the questions they ask; it is that they are asking the wrong questions. What is needed is an approach which can deal meaningfully with the question "How is language organized to convey meaning?" rather than "How are syntactic structures organized when viewed in isolation?".

How does a sentence convey meaning beyond the meanings of individual words? Here is the place for syntax. The structure of a sentence can be viewed as the result of a series of grammatical choices made in generating it. The speaker encodes meaning by choosing to build the sentence with certain "features", and the problem of the hearer is

to recognize the presence of those features and interpret their meaning.

We want to analyze the possible choices of features and functions which grammatical structures can have. For example, we might note that all sentences must be either IMPERATIVE, DECLARATIVE, or a QUESTION, and that in the last case they must choose as well between being a YES-NO question or a WH- question containing a word such as "why" or "which". We can study the way in which these features of sentences are organized -- which ones form mutually exclusive sets (called "systems"), and which sets depend on the presence of other features (like the set containing YES-NO and WH- depends on the presence of QUESTION). This can be done not only for full sentences, but for smaller syntactic units such as noun groups and prepositional groups, or even for individual words.

In addition we can study the different functions a syntactic "unit" can have as a part of a larger unit. In "Nobody wants to be alone.", the clause "to be alone" has the function of OBJECT in the sentence, while the noun group "nobody" is the SUBJECT. We can note that a transitive clause must have units to fill the functions of SUBJECT and OBJECT, or that a WH- question has to have some constituent which has the role of "question element" (like "why" in "Why did he go?" or "which dog" in "Which dog stole the show?").

In most current theories, these features and functions are implicit in the syntactic rules. There is no explicit mention of them, but the rules are designed in such a way that every sentence will in fact be one of the three types listed above, and every WH- question will in fact

have to have a question element. The difficulty is that there is no attempt in the grammar to distinguish significant features such as these from the infinite number of other features we could note about a sentence, and which are also implied by the rules.

If we look at the "deep structure" of a sentence, again the features and functions are implicit. The fact that it is a YES-NO question is indicated by a question marker hanging from a particular place in the tree, and the fact that a component is the object or subject is determined from its exact relation to the branches around it. The problem isn't that

there is no way to find these features in a parsing, but that most theories don't bother to ask "Which features of a syntactic structure are important to conveying meaning, and which are just a by-product of the symbol manipulations needed to produce the right word order."

What we would like is a theory in which these choices of features are primary. Professor M.A.K. Halliday at the University of London has been working on such a theory, called Systemic Grammar (see references (20), (21), (22), (26), and (27)). His theory recognizes that meaning is of prime importance to the way language is structured. Instead of having a "deep structure" which looks like a kind of syntactic structure tree, he deals with "system networks" which describe the way different features interact and depend on each other. The primary emphasis is on analyzing the limited and highly structured sets of choices which are made in producing a sentence or constituent. The exact way in which

these choices are "realized" in the final form is a necessary but secondary part of the theory.

The realization rules carry out the work which would be done by transformations in transformational grammar. In TG, the sentences "Sally saw the squirrel.", "The squirrel was seen by Sally.", and "Did Sally see the squirrel?" would all be derived from almost identical deep structures, and the difference in final form is produced by transformations. In systemic grammar, these would be analyzed as having most of their features in common, but differing in one particular choice, such as PASSIVE vs. ACTIVE, or DECLARATIVE vs. QUESTION. The realization rules would then describe the exact word order used to signal these features.

What does this theory give us to use in a language-understanding program? What kinds of parsings does it produce? If we look at a typical parsing by a systemic grammar, we note several points. First, it is very close to the surface structure of the sentence. There is no rearrangement into supposed "underlying" forms. Instead, each constituent is marked with features indicating its structure and function. Instead of saying that "Did John go?" has an underlying structure which looks like "John went.", we simply note that it has the features QUESTION and YES-NO, and that the noun group "John" has the function SUBJECT. Other parts of the language understanding process do not have to be concerned with the exact way the parsing tree is structured, since they can deal directly with the relevant features and

functions.

What is more important is that these features are not random unrelated lists of observations. They are part of a highly structured network, and the grammatical theory includes a description of that network. When we do semantic analysis, we are not faced with the task of inventing "projection rules" to deal with the specific syntactic rules. Instead we can ask "What aspect of meaning does this system convey?", and "What is the significance of this particular feature within its system?".

2.1.2 Parsing

In implementing a systemic grammar for a computer, we are concerned with the process of recognition rather than that of generation. We do not begin with choices of features and try to produce a sentence. Instead we are faced with a string of letters, and the job is to recognize the patterns and features in it. We need the inverse of realization rules -- interpretation rules which look at a pattern, identify its structure, and recognize its relevant features. This interpretation process is closely related to other types of pattern recognition, and many interesting parallels can be drawn with the process of interpreting a visual scene (see (50)). The important aspect of both types of interpretation is looking for symbolic features which will be relevant to understanding, so that the parsing can be integrated with the rest of the understanding process. In general, this problem of isolating important features from complex information and representing

them symbolically is a central issue for Artificial Intelligence, and the idea of a "systemic" parser may be of use in other areas.

The parsing system for our program is actually an interpreter for PROGRAMMAR, a language for writing grammars. It is basically a top-down left-to-right parser, but it modifies these properties when it is advantageous to do so. By writing in a language designed for grammars, we can express the regularities of language in a straightforward way, as simply as in a syntax-directed parser. By making it a programming language, we give the grammar power to use special tools to handle complex constructions and irregular forms.

For example, we can set up programs to define certain words like "and", and "or" as "demons", which cause an interrupt in the parsing process whenever they are found, in order to run a special program for conjoined structures. Idioms can also be handled using this "interrupt" concept.

It is paradoxical that linguistic workers familiar with computers have generally not appreciated the importance of the "control" aspect of programming, and have not used the process-describing potentialities of programming for their parsing theories. They have instead restricted themselves to the narrowest kinds of rules and transformations -- as though a programmer were to stick to such simple models as Turing machines or Post productions. Designers of computer languages today show this same tendency! See Minsky's remark in the April 1970 JACM (38). Our parser uses semantic guidance at all points, looking for a

meaningful parsing of the sentence rather than trying all of the syntactic possibilities. Section 2.2 describes PROGRAMMAR in detail, and 2.3 gives a sample grammar for English. Section 2.4 explains programming details, and shows how the special features of the language are actually used to handle specific linguistic problems.

2.2 A Description of PROGRAMMAR

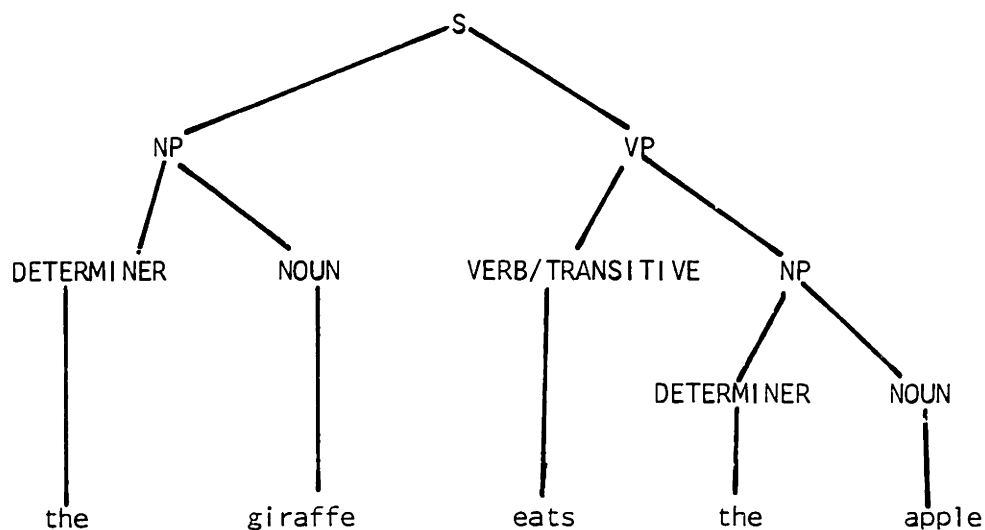
2.2.1 Grammar and Computers

In order to explain the features of PROGRAMMAR, we will summarize some of the principles of grammar used in computer language processing. The basic form of most grammars is a list (ordered or unordered) of "replacement rules," which represent a process of sentence generation. Each rule states that a certain string of symbols (its left side) can be replaced by a different set of symbols (its right side). These symbols include both the actual symbols of the language (called terminal symbols) and additional "non-terminal" symbols. One non-terminal symbol is designated as a starting symbol, and a string of terminal symbols is a sentence if and only if it can be derived from the starting symbol through successive application of the rules. For example we can write Grammar 1:

- 1.1 S -> NP VP
- 1.2 NP -> DETERMINER NOUN
- 1.3 VP -> VERB/INTRANSITIVE
- 1.4 VP -> VERB/TRANSITIVE NP
- 1.5 DETERMINER -> the
- 1.6 NOUN -> giraffe
- 1.7 NOUN -> apple
- 1.8 VERB/INTRANSITIVE -> dreams
- 1.9 VERB/TRANSITIVE -> eats

By starting with S and applying the list of rules (1.1 1.2 1.5 1.6 1.4 1.2 1.7 1.5 1.9), we get the sentence "The giraffe eats the apple."

Several things are noteworthy here. This is an unordered set of rules. Each rule can be applied any number of times at any point in the derivation where the symbol appears. In addition, each rule is optional. We could just as well have reversed the applications of 1.6 and 1.7 to get "The apple eats the giraffe.", or have used 1.3 and 1.8 to get "The giraffe dreams." This type of derivation can be represented graphically as:



We will call this the parsing tree for the sentence, and use the usual terminology for trees (node, subtree, daughter, parent, etc.). In addition we will use the linguistic terms "phrase" and "constituent" interchangeably to refer to a subtree. This tree represents the

"immediate constituent" structure of the sentence.

2.2.2 Context-free and Context-sensitive Grammars

Grammar 1 is an example of what is called a context-free grammar. The left side of each rule consists of a single symbol, and the indicated replacement can occur whenever that symbol is encountered. There are a great number of different forms of grammar which can be shown to be equivalent to this one, in that they can characterize the same languages. It has been pointed out that they are not theoretically capable of expressing the rules of English, to produce such sentences as, "John, Sidney, and Chan ordered an eggroll, a ham sandwich, and a bagel respectively." Much more important, even though they could theoretically handle the bulk of the English language, they cannot do this at all efficiently. Consider the simple problem of subject-verb agreement. We would like a grammar which generates "The giraffe dreams." and "The giraffes dream.", but not "The giraffe dream." or "The giraffes dreams.". In a context-free grammar, we can do this by introducing two starting symbols, S/PL and S/SG for plural and singular respectively, then duplicating each rule to match. For example, we would have:

- 1.1.1 S/PL -> NG/PL VP/PL
- 1.1.2 S/SG -> NG/SG VP/SG
- 1.2.1 NG/PL -> DETERMINER NOUN/PL
- 1.2.2 NG/SG -> DETERMINER NOUN/SG

...

1.6.1 NOUN/PL -> giraffes

1.6.2 NOUN/SG -> giraffe

etc.

If we then wish to handle the difference between "I am", "he is", etc. we must introduce an entire new set of symbols for first-person. This sort of duplication propagates multiplicatively through the grammar, and arises in all sorts of cases. For example, a question and the corresponding statement will have much in common concerning their subjects, objects verbs, etc., but in a context-free grammar, they will in general be expanded through two entirely different sets of symbols.

One way to avoid this problem is to use context-sensitive rules. In these, the left side may include several symbols, and the replacement occurs when that combination of symbols occurs in the string being generated.

2.2.3 Systemic Grammar

We can add power to our grammar with context-sensitive rules which, for example, in expanding the symbol VERB/INTRANSITIVE, look to the preceding symbol to decide whether it is singular or plural. By using such context-sensitive rules, we can characterize any language whose sentences can be listed by a deterministic (possibly neverending) process. (i.e. they have the power of a turing machine). There is however a problem in implementing these rules. In any but the simplest cases, the context will not be as obvious as in the simple example

given. The choice of replacements will not depend on a single word, but may depend in a complex way on the entire structure of the sentence. Such dependencies cannot be expressed in our simple rule format, and new types of rules must be developed. Transformational grammar solves this by breaking the generation process down into the context-free base grammar which produces "deep structure" and a set of transformations which then operate on this structure to produce the actual "surface structure" of the grammatical sentence. We will not go into the details of transformational grammar, but one basic idea is this separation of the complex aspects of language into a separate transformational phase of the generation process.

Systemic grammar introduces context in a more unified way into the immediate-constituent generation rules. This is done by introducing "features" associated with constituents at every level of the parsing tree.

A rule of the grammar may depend, for example, on whether a particular clause is transitive or intransitive. In the examples "Fred found a frog.", "A frog was found by fred.", and "What did fred find?", all are transitive, but the outward forms are quite different. A context-sensitive rule which checked for this feature directly in the string being generated would have to be quite complex. Instead, we can allow each symbol to have additional subscripts, or features which control its expansion. In a way, this is like the separation of the symbol NP into NP/PL and NP/SG in our augmented context-free grammar. But it is not

necessary to develop whole new sets of symbols with a set of expansions for each. A symbol such as CLAUSE may be associated with a whole set of features (such as TRANSITIVE, QUESTION, SUBJUNCTIVE, OBJECT-QUESTION, etc.) but there is a single set of rules for expanding CLAUSE. These rules may at various points depend on the set of features present.

The power of systemic grammar rests on the observation that the context-dependency of natural language is centered around clearly defined and highly structured sets of features, so through their use a great deal of complexity can be handled very economically. More important for our purposes, there is a high correlation between these features and the semantic interpretation of the constituents which exhibit them. They are not directly semantic, but are a tremendous aid to interpretation. A parsing of a sentence in a systemic grammar might look very much like a context-free parsing tree, except that to each node would be attached a number of features.

These features are not random combinations of facts about the constituent, but are a part of a carefully worked out analysis of a language in terms of its "systems". The features are organized in a network, with clearly organized dependencies. For example, the features IMPERATIVE (command) and QUESTION are mutually exclusive in a clause, as are the features YES-NO (yes-no question like "Did he go?") and WH-question (like "Who went?"). In addition, the second choice can be made only if the choice QUESTION was made in the first set. A set of mutually exclusive features is called a "system", and the set of other

features which must be present for the choice to be possible is called the "entry condition" for that system. This is discussed in detail in section 2.3.

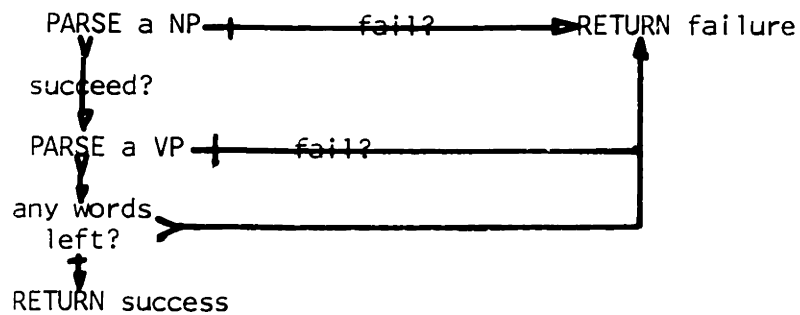
Another basic concept of systemic grammar is that of the rank of a constituent. Rather than having a plethora of different non-terminal symbols, each expanding a constituent in a slightly different way, there are only a few basic "units", each having the possibility of a number of different features, chosen from the "system network" for that unit. In an analysis of English, three basic units seem to explain the structure: the CLAUSE, the GROUP, and the WORD. In general, clauses are made up of groups, and groups made up of words. However, through "rankshift", clauses or groups can serve as constituents of other clauses or groups. Thus, in the sentence "Sarah saw the student sawing logs." "the student sawing logs" is a NOUN GROUP with the CLAUSE "sawing logs" as a constituent (a modifier of "student").

The constituents "who", "three days", "some of the men on the board of directors," and "anyone who doesn't understand me" are all noun groups, exhibiting different features. This means that a PROGRAMMAR grammar will have only a few programs, one to deal with each of the basic units. Our current grammar of English has programs for the units CLAUSE, NOUN GROUP, VERB GROUP, PREPOSITION GROUP, and ADJECTIVE GROUP.

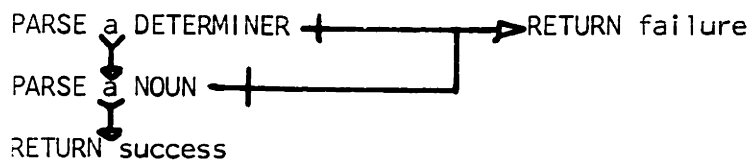
2.2.4 Grammars as Programs

Let us see how a grammar as outlined above could be written as a program. Grammar 1 could be diagrammed:

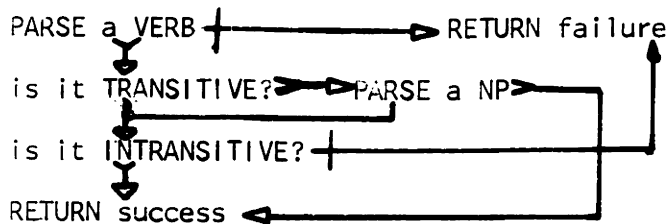
DEFINE program SENTENCE



DEFINE program NP



DEFINE program VP



The basic function used is PARSE, a function which tries to add a constituent of the specified type to the parsing tree. If the type has been defined as a PROGRAMMAR program, PARSE activates the program for that unit, giving it as input the part of the sentence yet to be parsed. If no definition exists, PARSE interprets its arguments as a list of

features which must be found in the dictionary definition of the next word in the sentence. If so, it attaches a node for that word, and removes it from the remainder of the sentence. If not, it fails. If a PPROGRAMMAP program has been called and succeeds, the new node is attached to the parsing tree. If it fails, the tree is left unchanged.

2.2.5 The Form of PPROGRAMMAP Grammars

Written in PPROGRAMMAP, the programs would look like:

```

2.1 (PDEFINE SENTENCE
2.2 (((PARSE NP) NIL FAIL)
2.3  ((PARSE VP) FAIL FAIL RETURN)))

2.4 (PDEFINE NP
2.5 (((PARSE DETERMINER) NIL FAIL)
2.6  ((PARSE NOUN) RETURN FAIL)))

2.7 (PDEFINE VP
2.8 (((PARSE VEB) NIL FAIL)
2.9  ((ISO H TRANSITIVE) NIL INTRANS )
2.10 ((PARSE NP) RETURN NIL)
2.11 INTRANS
2.12 ((ISO H INTRANSITIVE) RETURN FAIL)))

```

Rules 1.6 to 1.9 would have the form:

```

2.13 (DEFPROP GIRAFFE (NOUN) WORD)
2.14 (DEFPROP DREAM (VEB INTRANSITIVE) WORD)
etc.

```

This example illustrates some of the basic features of PPROGRAMMAP. First it is embedded in LISP, and much of its syntax is LISP syntax. Units, such as SENTENCE are defined as PPROGRAMMAP programs of no arguments. Each tries to parse

the string of words left to be parsed in the sentence. The exact form of this input string is described in section 2.4.3. The value of (PARSE SENTENCE) will be a list structure corresponding to the parsing tree for the complete sentence.

Each time a call is made to the function PARSE, the system begins to build a new node on the tree. Since PROGPANMAP programs can call each other recursively, it is necessary to keep a pushdown list of nodes which are not yet completed (i.e. the entire rightmost branch of the tree). These are all called "active" nodes, and the one formed by the most recent call to PARSE is called the "currently active node".

We can examine our sample program to see the basic operation of the language. Whenever a PROGPANMAP program is called directly by the user, a node of the tree structure is set up, and a set of special variables are bound (see section 2.4.6). The lines of the program are then executed in sequence, as in a LISP PROG, except when they have the special form of a BRANCH statement, a list whose first member (the CONDITION) is non-atomic, and which has either 2 or 3 other members, called DIRECTIONS. Line 2.3 is a three-direction branch, and all the other executable lines of the program are two-direction branches.

When a branch statement is encountered, the condition is evaluated, and branching depends on its value. In a two-direction branch, the first direction is taken if it evaluates to non-nil, the second direction if it is nil. In a three-direction branch, the first direction is taken only if the condition is non-nil, and there is more of the sentence to be parsed. If no more of the sentence remains, the third direction is taken.

The directions can be of three types. First, there are three reserved words, NIL, PETUPN, and FAIL. A direction of NIL sends evaluation to the next statement in the program. FAIL causes the program to return NIL after restoring the sentence and the parsing tree to their state before that program was called. PETUPN causes the program to attach the currently active node to the completed parsing tree and return the subtree below that node as its value.

If the direction is any other atom, it acts as a GO statement, transferring evaluation to the statement immediately following the occurrence of that atom as a tag. For example, if a failure occurs in line 2.9, evaluation continues with line 2.12. If the direction is non-atomic, the result is the same as a FAIL, but the direction is put on a special failure message list, so the calling program can see the reason for failure.



su
us
th
ne
re
se
tl
to
fe
ho
(s
co
H
ha
it
fi
or
IN
su
IN
de
2.

grammar. How, for example, can we handle agreement? One way to do this would be for the VP program to look back in the sentence for the subject, and check its agreement with the verb before going on. We need a way to climb around on the parsing tree, looking at its structure. In PPOGPANMAP, this is done with the pointer PT and the moving function *.

Whenever the function * is called, its arguments form a list of instructions for moving PT from its present position. The instruction list contains non-atomic CONDITIONS and atomic INSTRUCTIONS. The instructions are taken in order, and when a condition is encountered, the preceding instruction is evaluated repeatedly until the condition is satisfied. If the condition is of the form (ATOM), it is satisfied only if the node pointed to by PT has the feature ATOM. Any other condition is evaluated by LISP, and is satisfied if it returns a non-nil value. Section 2.4.8 lists the instructions for *.

For example, evaluating (* C U) will set the pointer to the parent of the currently active node. (The mnemonics are: Current, Up) The call (* C DLC PV (NP)) will start at the current node, move down to the rightmost completed node (i.e. not currently active) then move left until it finds a node with the feature NP. (Down-Last-Completed, PreVIOUS). If * succeeds, it returns the new value of PT and leaves PT

set to that value. If it fails at any point in the list, because the existing tree structure makes a command impossible, or because a condition cannot be satisfied, PT is left at its original position, and * returns nil.

We can now add another branch statement to the VP program in section 2.2.5 between lines 2.8 and 2.9 as follows:

```

2.8.1 ((OR(AND(ISO(* C PV DLC)SINGULAR)(ISO H SINGULAR))
2.8.2      (AND(ISO PT PLURAL)(ISO H PLURAL)))
2.8.3  NIL (AGREEMENT))

```

This is an example of a branch statement with an error message. It moves the pointer from the currently active node (the VP) to the previous node (the NP) and down to its last constituent (the noun). It then checks to see whether this shares the feature SINGULAR with the last constituent parsed by VP (the verb). If not it checks to see whether they share the feature PLURAL. Notice that once PT has been set by *, it remains at that position. If agreement is found, evaluation continues as before with line 2.9. If not, the program VP fails with the message (AGREEMENT).

So far we have not made much use of features, except on words. As the grammar gets more complex, they become much more important. As a simple example, we may wish to augment our grammar to accept the noun groups

"these fish,"
 "this fish,"
 "the giraffes,"
 and "the giraffe,"

But not "these giraffe," or "this giraffes." We can no longer check a single word for agreement, since "fish" gives no clue to number in the first two, while "the" gives no clue in the third and fourth. Number is a feature of the entire noun group, and we must interpret it in some cases from the form of the noun, and in others from the form of the determiner.

We can rewrite our programs to handle this complexity as shown in Grammar 3:

```

3.1 (PDEFINE SENTENCE
3.2 (((PARSE NP)NIL FAIL)
3.3 ((PARSE VP) FAIL FAIL RETURN)))

3.4 (PDEFINE NP
3.5 (((AND(PARSE DETERMINER)(FO DETERMINED))NIL NIL FAIL)
3.6 ((PARSE NOUN)NIL FAIL)
3.7 ((CO DETERMINED)DET NIL)
3.8 ((TRNSF H (QUOTE(SINGULAR PLURAL)))RETURN FAIL)
3.9 DET
3.10 ((TRNSF H (MEET(FE(* H PV (DETERMINER)))
3.11 (QUOTE(SINGULAR PLURAL))))
3.12 RETURN
3.13 FAIL)))

```

```

3.14 (PDEFINE VP
3.15 ((PARSE VERR)NIL FAIL)
3.16 ((MEET(FE H)(FE(* C PV (NP)))(QUOTE(SINGULAR PLURAL)))
3.17  NIL
3.18  (AGREEMENT))
3.19 ((ISO H TRANSITIVE)NIL INTPANS)
3.20 ((PARSE NP)PETHUPN NIL)
3.21 ((ISO H INTPANSITIVE)PETHUPN FAIL)))

```

We have used the PPOGPAM'AP functions FO and TPNSF, which attach features to constituents. The effect of evaluating (FO A) is to add the feature A to the list of features for the currently active node of the parsing tree. TPNSF is used to transfer features from another node to the currently active node. Its first argument is a pointer to the node from which information is to be transferred. The second is a list of features to be looked for. For example, line 3.8 looks for the features SINGULAR and PLURAL in the last constituent parsed (the NOUN), and adds whichever ones it finds to the currently active node. The branch statement beginning with line 3.10 is more complex. The function * finds the DETERMINER of the NP being parsed. The function FE finds the list of features of this node, and the function MEET intersects this with the list of features (SINGULAR PLURAL). This intersection is then the set of allowable features to be transferred to the NP node from the NOUN. Therefore if there is no agreement between the NOUN and the DETERMINER, TRNSF fails to find any features to

transfer, and the resulting failure causes the rejection of such phrases as "these giraffe."

In line 3.7 we use the function CO which checks for features on the current node. (CO DETERMINED) will be non-nil only if the current node has the feature DETERMINED. (i.e. it was put there in line 3.5) Therefore, a noun group with a determiner is marked with the feature DETERMINED, and is also given features corresponding to the intersection of the number features associated with the determiner if there is one, and the noun. Notice that this grammar can accept noun groups without determiners, as in "Giraffes eat apples." since line 3.5 fails only if a DETERMINED is found and there are no more words in the sentence.

In conjunction with the change to the NP program, the VP program must be modified to check with the NP for agreement.

The branch statement beginning on Line 3.16 does this by making sure there is a number feature common to both the subject and the verb.

This brief description explains some of the basic features of PROGRAMAP. In a simple grammar, their importance is not obvious, and indeed there seem to be easier ways to achieve the same effect. As grammars become more complex, the special aspects of PROGRAMAP become more

and more important. The flexibility of writing a grammar as a program is needed both to handle the complexities of English syntax, and to combine the semantic analysis of language with the syntactic analysis in an intimate way. Section 2.3 describes a fairly complex grammar of English, and section 4.2 describes the way it is integrated with the semantic programs. A number of the other features and details of PROGRAMMAP are described in Section 2.4.

2.2.7 Ambiguity and Understanding

Readers familiar with parsing systems may by now have wondered about the problem of ambiguity. As explained, a PROGRAMMAP program tries to find a possible parsing for a sentence, and as soon as it succeeds, it returns its answer. This is not a defect of the system, but an active part of the concept of language for which it was designed. The language process is not segmented into the operation of a parser, followed by the operation of a semantic interpreter. Rather, the process is unified, with the results of semantic interpretation being used to guide the parsing. This is very difficult in other forms of grammar, with their restricted types of context-dependence. But it is straightforward to implement in PROGRAMMAP. For example, the last statement in a program for NP may be a call to a noun-phrase semantic interpreter. If it is impossible to

interpret the phrase as it is found, the parsing is immediately redirected.

The way of treating ambiguity is not through listing all 1,243 possible interpretations of a sentence, but in being intelligent in looking for the first one, and being even more intelligent in looking for the next one if that fails. There is no automatic backup mechanism in PROGRAMMAR, because blind automatic backup is tremendously inefficient. A good PROGRAMMAR program will check itself when a failure occurs, and based on the structures it has seen and the reasons for the failure, it will decide specifically what should be tried next. This is the reason for internal failure-messages, and there are facilities for performing the specific backup steps necessary. (See section 2.4.4)

As a concrete example, we might have the sentence "I rode down the street in a car." At a certain point in the parsing, the NP program may come up with the constituent "the street in a car". Before going on, the semantic analyzer will reject the phrase "in a car" as a possible modifier of "street", and the program will attach it instead as a modifier of the action represented by the sentence. Since the semantic system is a part of a complete deductive understander, with a definite world-model, the semantic

evaluation which guides parsing can include both general knowledge (cars don't contain streets) and specific knowledge (Melvin owns a red car, for example). Humans take advantage of this sort of knowledge in their understanding of language, and it has been pointed out by a number of linguists and computer scientists that good computer handling of language will not be possible unless computers can do so as well.

Very few sentences seem ambiguous to humans when they first hear them. They are guided by an understanding of what is said to pick a single parsing and a very few different meanings. By using this same knowledge to guide its parsing, a computer understanding system could take advantage of the same technique to parse meaningful sentences quickly and efficiently. We must be careful to distinguish between grammatical and semantic ambiguity. Although we want to choose a single parsing without considering the alternatives simultaneously, we want to handle semantic ambiguity very differently. There may be several interpretations of a sentence which are all more or less meaningful, and the choice between them will depend on a complex evaluation of our knowledge of the world, of the knowledge the person speaking has of the world, and of what has been said recently. This is particularly true in cases

of ambiguous pronoun reference. For example, if the system were asked the sequence of questions: "Is a green block on a table?" "What color is it?" we would expect "it" to refer to the table. If asked "Is a block on a green table? "What color is it?", we know that "it" must refer to the block. If the second question were "What size is it?" it would be much more ambiguous. To resolve this, we must know that green is a color, and that a person is not likely to ask the color of an object if he has just specified it. This is the type of ambiguity dealt with by the system through its heuristic programs and deductive system. It however is not directly a part of the grammar.

This means that we must discuss only those things the computer knows about and understands fully enough to manipulate the knowledge necessary for interpreting a sentence. The forms of relationships it must be able to handle go beyond simple structures, such as associative networks of links between words or arbitrary relational statements in the predicate calculus. Other sections of this paper describe the ways we include this type of knowledge in the language-understanding program.

2.2.8 Summary

In understanding the reason for developing PROGRAMAP, several factors are important. The first is that only

through the flexibility of expressing a grammar as a program can we introduce the type of intelligence necessary for complete language understanding. PPROGRAMMAP is able to take into account the fact that language is structured in order to convey meaning, and that our parsing of sentences depends intimately on our understanding that meaning. PPROGRAMMAP can take advantage of this to deal more efficiently with natural language than a general rule-based system, whether context-free or transformational. More important, the analysis returned by PPROGRAMMAP is designed to serve as a part of a total understanding process, and to lend itself directly to semantic interpretation. This was one reason for selecting systemic grammar, and has guided much of the design of the system. The exact way in which semantic interpretation can be done, and the reasons why a systemic analysis is important will be discussed in sections 4.1 and 4.2.

2.3 A Grammar of English

2.3.1 About the Grammar

This section describes the grammar of English used by our system. It is based on the principles of systemic grammar (Halliday (20), (21), (22)), and emphasizes the analysis of the significant features exhibited by linguistic structures, rather than their detailed form. Instead of giving a set of grammar rules or algorithms, this section will try to convey an impressionistic overview of English grammar, giving examples of the most important features. Section 2.3.9 follows the parser through two sample sentences, showing how the grammar is used. The actual PROGRAMMAP grammar contains the details, and is available on request. A few particularly interesting parts of the detailed grammar are described in section 2.4.

Before beginning, several warnings are in order. First, this is not by any means a complete grammar of English. The task of codifying an entire language in any formalism is so large that it would be folly to try in the course of a single research project. Our goal was to cover a large enough portion of English syntax so that a user could converse comfortably with the system about its toy-block world. There are whole areas of syntax which are involved with conveying information of types not included in

this narrow field (such as the emotional reaction, mood, and emphasis of the speaker). These are not handled at all, and even within the toy-block world, there are numerous sentences and constructions which the grammar is not yet equipped to handle. It will be of interest to see whether the basic structure of the syntactic theory is flexible enough to add the great amount of complexity which could be included in a more complete grammar.

Second, the grammatical theory is used in a very impure way. The main consideration was to produce a working grammar which could serve in a language-understanding program. The demands of practicality often overrode more theoretical criteria, and the resulting grammar is not very "pretty". This is especially true since it has evolved in a continuous process of writing and debugging, and has not yet undergone the "polishing" which removes the traces of its earlier stages of development.

Demands of time made it impossible to coordinate the writing of the grammar with other current versions of systemic grammar, so the analysis is non-standard, often disagreeing with Halliday's analysis or other more complete versions. Some differences are simply notational (using different names for the same thing), others are intentional simplifications (Halliday's analysis is much more complete),

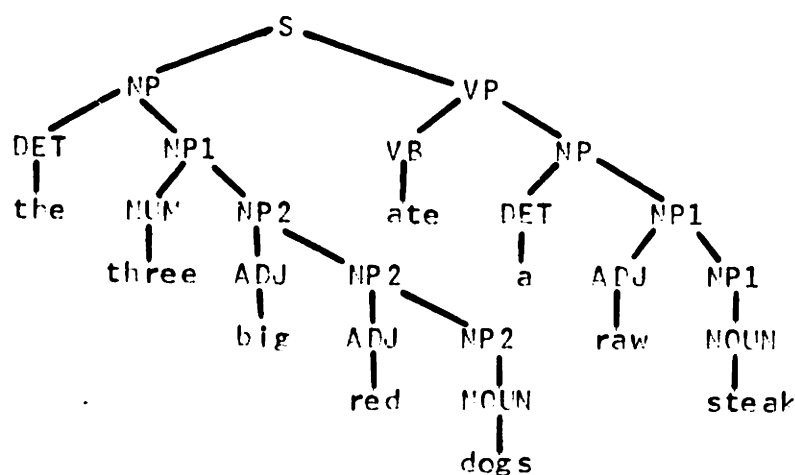
and some represent actual theoretical differences (for example, our analysis of the transitivity system puts much of the structure into the semantic rather than syntactic rules, while Halliday's is more purely syntactic.). We will not describe the differences in detail, since this is not a proposal for a specific version of English grammar. It is instead a proposal for a way of looking at language, and at English, pointing out some of the interesting features.

2.3.2 Units, Rank, and Features

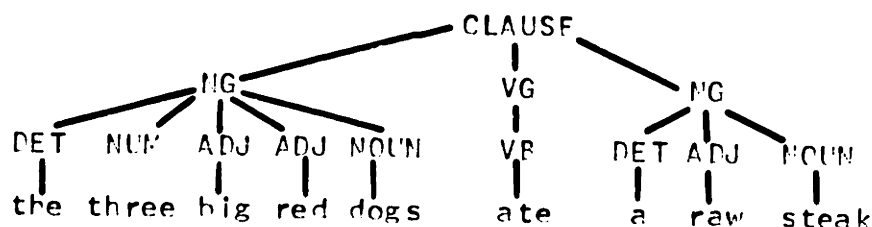
We will begin by describing some of the basic concepts of systemic grammar, before giving details of their use in our analysis. Some of the description is a repetition of material in Section 2.1. In that section we needed to give enough explanation of systemic grammar to explain PROGRAMMAR. Here we give a more thorough explanation of its details.

The first is the notion of syntactic units in analyzing the constituent structure of a sentence (the way it is built up out of smaller parts). If we look at other forms of grammar, we see that syntactic structures are usually represented as a binary tree, with many levels of branching and few branches at any node. The tree is not organized into "groupings" of phrases which are used for conveying different parts of the meaning. For example, the sentence

"The three big red dogs ate a raw steak." would be parsed with something like:



Systemic grammar pays more attention to the way language is organized into units, each of which has a special role in conveying meaning. In English we can distinguish three basic ranks of units, the CLAUSE, the GROUP, and the WORD. There are several types of groups: NOUN GROUP (NG), VERB GROUP (VG), PREPOSITION GROUP (PPEPG) and ADJECTIVE GROUP (ADJG). In a systemic grammar, the same sentence might be viewed as having the structure:



In this analysis, the WORD is the basic building block. There are word classes like "adjective", "noun", "verb", and each word is an integral unit -- it is not chopped into hypothetical bits (like analyzing "dogs" as being composed of "dog" and "-s" or "dog" and "plural"). Instead we view each word as exhibiting features. The word "dogs" is the same basic vocabulary item as "dog", but has the feature "plural" instead of "singular". The words "took", "take", "taken", "taking", etc., are all the same basic word, but with differing features such as "past participle" (EN), "infinitive" (INF), "-ing" (ING), etc. When discussing features, we will use several notational conventions. Any word appearing in all upper-case letters, is the actual symbol used to represent a feature in our grammar and semantic programs. A feature name enclosed in quotes is an English version which is more informative. Usually the program version is an abbreviation of the English version, and sometimes we will indicate this by typing the letters of the abbreviation in upper-case, and the rest in lower-case. Thus if "determiner" is abbreviated as DET, we may write

DETerminer. We may even write things like QuaNTIFieP. When we want to be more careful, we will write "quantifier" (QNTFR).

The next larger unit than the WORD is the GROUP, of which there are the four types mentioned above. Each one has a particular function in conveying meaning. Noun groups (NG) describe objects, verb groups (VG) carry complex messages about the time and modal (logical) status of an event or relationship, preposition groups (PPEPG) describe simple relationships, while adjective groups (ADJG) convey other kinds of relationships and descriptions of objects. These semantic functions are described in more detail in section 4.2.

Each GROUP can have "slots" for the words of which it is composed. For example, a NG has slots for things like "determiner" (DET), "numbers" (NUM), "adjectives" (ADJ), "classifiers" (CLASF), and a NOUN. Each group can also exhibit features, just as a word can. A NG can be "singular" (NS) or "plural" (NPL), "definite" (DEF) as in "the three dogs" or "indefinite" (INDEF) as in "a steak", and so forth. A VG can be "negative" (NEG) or not, can be MODAL (as in "could have seen"), and it has a tense. (See below for an analysis of complicated tenses, such as "He would have been going to be fixing it.")

Finally, the top rank is the CLAUSE. We speak of clauses rather than sentences since the sentence is more a unit of discourse and semantics than a separate syntactic structure. It is either a single clause or a series of clauses joined together in a simple structure such as "A and B and...". We study these conjoining structures separately since they occur at all ranks, and there is no real need to have a separate syntactic unit for sentence.

The clause is the most complex and diverse unit of the language, and is used to express complex relationships and events, involving time, place, manner and many other aspects of meaning. It can be a QUESTION, a DECLARATIVE, or an IMPEPATIVE, it can be "passive" (PASV) or "active" (ACTV), it can be a YES-NO question or a WH- question (like "Why...?" or "Which...?").

Looking at our sample parsing tree, we see that the clauses are made up of groups, which are in turn made up of words. However few sentences have this simple three-layer structure. Groups often contain other groups (for example, "the call of the wild" is a NG, which contains the PPEPG "of the wild" which in turn contains the NG "the wild"). Clauses can be parts of other clauses (as in "Join the Navy to see the world."), and can be used as parts of groups in many different ways (for example, in the NG "the man who

came to dinner" or the PPEPG "by leaving the country".) This phenomenon is called rankshift, and is one of the basic principles of systemic grammar.

If the units can appear anywhere in the tree, what is the advantage of grouping constituents into "units" instead of having a detailed structure like the one shown in our first parsing tree? The answer is in the "features" we were noting above. Each unit has associated with it a set of features, which are of primary significance in conveying meaning. We mentioned that a clause could have features such as IMPERATIVE, DECLARATIVE, QUESTION, ACTV, PASV, YES-NO, and WH-. These are not unrelated observations we can make about a clause. They are related by a definite logical structure. The choice between YES-NO and WH- is meaningless unless the clause is a QUESTION, but if it is a QUESTION, the choice must be made. Similarly, the choice between QUESTION, IMPERATIVE, and DECLARATIVE is mandatory for a MAJOR clause (one which could stand alone as a sentence), but is not possible for a "secondary" (SEC) clause, such as "the country which possesses the bomb." The choice between PASV (as in "the ball was attended by John",) and ACTV (as in "John attended the ball.") is on a totally different dimension, since it can be made regardless of which of these other features are present.

We can represent these logical relationships graphically using a few simple conventions. A set of mutually exclusive features (such as QUESTION, DECLARATIVE, and IMPERATIVE) is called a system, and is represented by connecting the features with a vertical bar:

```

!QUESTION
!
!DECLARATIVE
!
!IMPERATIVE

```

The vertical order is not important, since a system is a set of unordered features among which we will choose one. Each system has an entry condition which must be satisfied in order for the choice to be meaningful. This entry condition can be an arbitrary boolean condition on the presence of other features. The simplest case (and most common) is the presence of a single other feature. For example, the system just depicted has the feature MAJOR as its entry condition, since only MAJOR clauses make the choice between DECLARATIVE, IMPERATIVE, and QUESTION.

This simple entry condition is represented by a horizontal line, with the condition on the left of the system being entered. We can diagram some of our CLAUSE features as:

```

                                !DECLAPATIVE
                                !
                                !MAJOP---!IMPEPATIVE
                                !
    CLAUSE---!                   !QUESTION-----!
                                !
                                !YES-NO
                                !
                                !WH-
  
```

Often there are independent systems of choices sharing the same entry condition. For example, the choice between SEC and MAJOP and the choice between PASV and ACTV both depend directly on the presence of CLAUSE. In standard notation, this is represented by a bracket, but for ease of typing, we will use a vertical bar with a "0" at the top and bottom, and with a blank space following the bar:

```

    0          !MAJOP---...
    ! -----!
    !         !SEC
    CLAUSE!
    !         !PASV
    ! -----!
    0          !ACTV
  
```

If we want to assign a name to a system (to talk about it), we can put the name above the line leading into it:

```

    VOICE !PASV
    -----!
           !ACTV
  
```

We can look at these notations as representing the logical operations of "or" and "and", and we can use them to represent more complex entry conditions. If the choice between the features C and D depends on the presence of either A or B, we draw;

```

A-----!      !C
          !-----!
B-----!      !C

```

and if the entry condition for the "C-D" system is the presence of both A and B, we write:

```

      0
A---- !      !C
      !-----!
B---- !      !D
      0

```

Finally, we can allow "unmarked" features, in cases where the choice is between the presence or absence of something of interest. We might have a system like;

```

NEGATIVITY      !NEGATIVE
-----!
              !---

```

in which the feature "non-negative" is not given a name, but is assumed unless the feature NEGATIVE is present.

We will explain our grammar by presenting the system networks for all three ranks -- CLAUSE, GROUP, and WORD, and giving examples of sentences exhibiting the features. We have not attempted to show all of the logical relationships in the networks -- our networks may indicate combinations of features which are actually not possible, and would need a more complex network to represent properly. We have chosen clarity over completeness whenever there was a conflict. In addition, we have represented "features" of units (i.e. descriptions of their structure) and "functions"

(descriptions of their use) in the same network. In a more theoretical presentation, it would be preferable to distinguish the two. The names chosen for features were arbitrary mnemonics invented as they were needed, and are neither as clear nor as systematic as they might be in a "cleaned up" version.

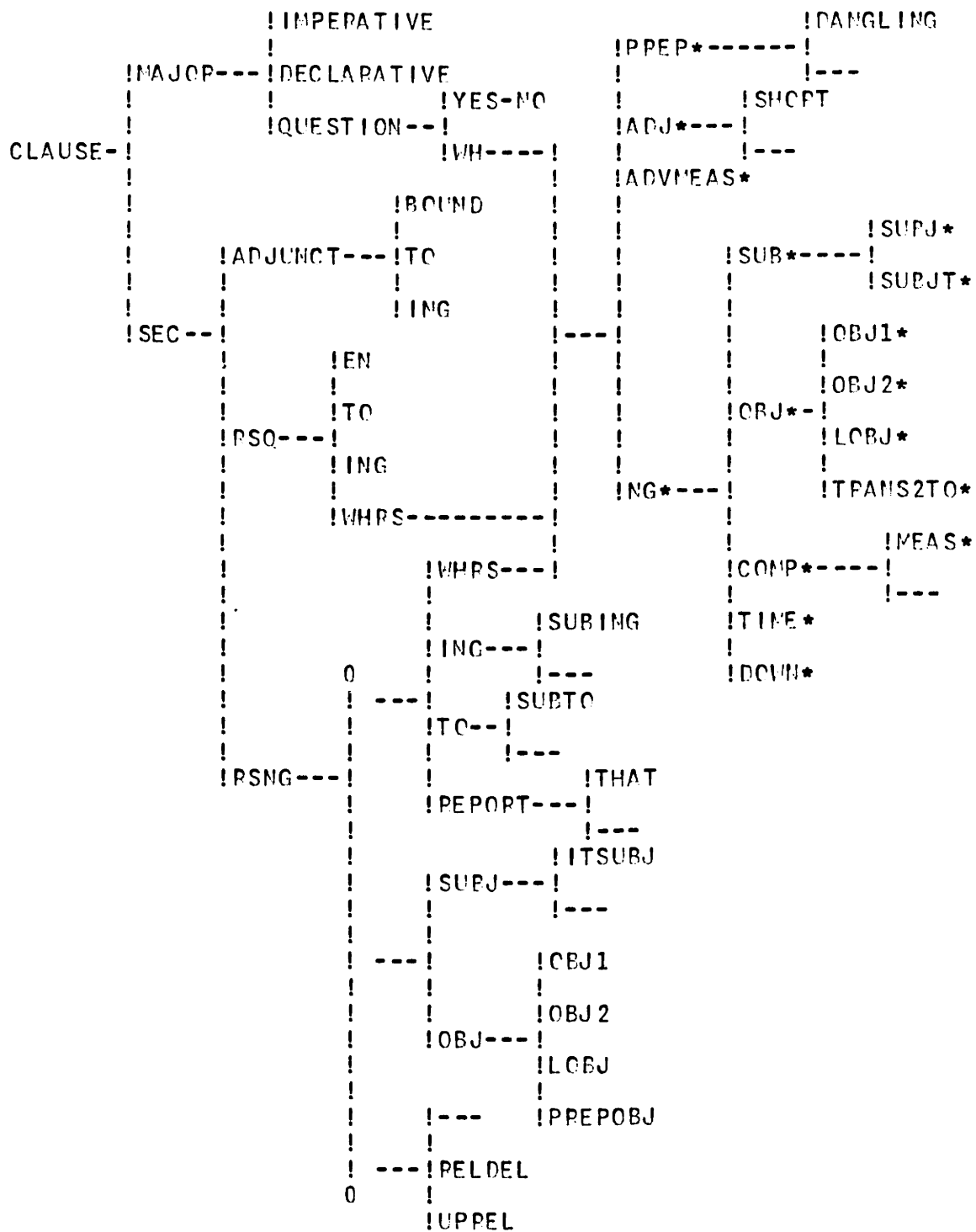
2.3.3 The CLAUSE

The structure exhibiting the greatest variety in English is the CLAUSE. It can express relationships and events involving time, place, manner, and other modifiers. Its structure indicates what parts of the sentence the speaker wants to emphasize, and can express various kinds of focus of attention and emotion. It determines the purpose of an utterance -- whether it is a question, command, or statement -- and is the basic unit of discourse which can stand alone. Other units can occur by themselves when their purpose is understood, as in answer to a question, but the clause is the only one which can stand alone without regard to discourse.

The CLAUSE has several main ingredients and a number of optional ones. Except for special types of incomplete clauses, there is always a verb group, containing the verb, which indicates the basic event or relationship being expressed by the CLAUSE. Almost every CLAUSE contains a

subject, except for IMPERATIVE (in which the semantic subject is understood to be the person being addressed), and embedded clauses in which the subject lies somewhere else in the syntactic structure. In addition to the subject, a CLAUSE may have various kinds of objects, which will be explained in detail later. It can take many types of modifiers (CLAUSES, GROUPS, and WORDS) which indicate time, place, manner, causality, and a variety of other aspects of meaning.

Let us look at one part of the CLAUSE system network:



NETWORK 1

Beginning at the top of the network, we see a choice between MAJOR (a clause which could stand alone as a sentence) and "secondary" (SEC). A MAJOR clause is either an IMPERATIVE (a command), a DECLARATIVE, or a QUESTION. Questions are either YES-NO -- answerable by "yes" or "no", as in:

(s1) Did you like the show?

or WH- (involving a question element like "when", "where", "which", "how", etc.). The choice of the WH- feature leads into a whole network of further choices, which are shared by QUESTION and two kinds of SECONdary clauses we will discuss later. In order to share the network, we have used a simple notational trick -- the symbols contain a "*", and when they are being applied to a question, we replace the * with "Q", while when they are applied to relative clauses, we use "REL". For example, the feature "PPEP*" in the network will be referred to as PREPQ when we find it in a question, but PPEPREL when it is in a relative clause. This is due to the way the grammar evolved, and in later versions we will probably use only one name for these features. This complex of features is basically the choice of what element of the sentence is being questioned. English allows us to use almost any part of a clause as a request for information. For example, in a PREPQ, a prepositional group

in the clause is used, as in:

(s2) With what did you erase it?

We more commonly find the preposition in a DANGLING position, as in:

(s3) What did you erase it with?

We can tell by tracing back through Network 1 that sentence s3 has the features PPEPQ, DANGLING, VH-, QUFSTION, and MAJOR.

We can use a special question adverb to ask questions of time, place, and manner, as in:

(s4) Why did the chicken cross the road?

(s5) When were you born?

(s6) How will you tell her the news?

(s7) Where has my little dog gone?

These are all marked by the feature QADJ. In discourse they can also appear in a short form (SHOPTO) in which the entire utterance is a single word, as in:

(s8) Why?

We can use the word "how" in connection with a measure adverb (like "fast") to ask an ADVMEASO, like:

(s9) How fast can he run the mile?

The most flexible type of VH- question uses an entire noun group as the question element, using a special pronoun (like "what" or "who") or a determiner (like "which", or "how many") to indicate that it is the question element.

These clauses have the feature NGO, and they can be further divided according to the function of the NG in the clause. It can have any of the possible NG functions (these will be described more formally with regard to the next network).

For example, it can be the subject, giving a SUBJO, like:

(s10) Which hand holds the M and M's?

It can be the subject of a THERE clause (see below), giving us a SUBJTO:

(s11) How many Puerto Ricans are there in Boston?

A complement is the second half of an "is" clause, like

:

(s12) Her hair is red.

and it can be used to form a COMPQ:

(s13) What color was her hair?

or with a "measure" in a MEASQ:

(s14) How deep is the ocean?

The noun group can be an object, leading to the feature OBJQ, as in:

(s15) What do you want? or

(s16) Who did you give the book?

These are both OBJQ, since the first has only one object ("what"), and the second questions the first, rather than the second object ("who", instead of "the book"). We use the ordering of the DECLAPATIVE form "You gave me the

book". If this were reversed, we would have an OBJ20,
like:

(s17) What did you give him?

If we use the word "to" to express the first object with
a two object verb like "give", we can get a TRANSTO20, like:

(s18) To whom did you give the book? or

(s19) Who did you give the book to?

Sometimes a NG can be used to indicate the time in a
clause, giving us a TIME0:

(s20) What day will the iceman come?

In a more complex style, we can embed the question
element within an embedded clause, such as:

(s21) Which car did your brother say that
he was expecting us to tell Jane to buy?

The NG "which car" is the question element, but is in
fact the object of the clause "Jane to buy...", which is
embedded several layers deep. This kind of NGO is called
DOWN0. The role of the question element in the embedded
clause can include any of those which we have been
describing. For example it could be the object of a
preposition, as in

(s22) What state did you say Lincoln was born in?

Looking at the network for the features of SEcONdary
clauses, we see three main types -- ADJUNCT, "Pank-Shifted
Qualifier" (PSQ), and "Pank-Shifted to act as a Noun Group"

(PSNG). ADJUNCT clauses are used as modifiers to other clauses, giving time references, causal relationships, and other similar information. We can use a ROUND clause containing a "binder" such as "before", "while", "because", "if", "so", "unless", etc., as in:

- (s23) While Nero fiddled, Rome burned.
- (s24) If it rains, stay home.
- (s25) Is the sky blue because it is cold?

To express manner and purpose, we use a TO clause or an ING clause:

- (s26) He died to save us from our sins.
- (s27) The bridge was built using primitive tools.

The PSQ clause is a constituent of a NG, following the noun in the "qualifier" position (see Section 2.3.4 for a description of the positions in a NG). It is one of the most commonly used secondary clauses, and can be of four different types. Three of them are classified by the form of the verb group within the clause -- TO, ING, and EN (where we use "en" to represent a past participle, such as "broken"):

- (s28) the man to see about a job
- (s29) the piece holding the door on
- (s30) a face weathered by sun and wind

Notice that the noun being modified can have various roles in the clause. In examples 28 and 29, "piece" is the subject of "hold", while "man" is the object of "see". We

could have said:

(s31) the man to do the job

in which "man" is the subject of "do". Our semantic analysis sorts out these possibilities in determining the meaning of a secondary clause.

The fourth type of RSO clause is related to WH- questions, and is called a WHRS. It uses a wh- element like "which" or "what", or a word like "that" to relate the clause to the noun it is modifying. The different ways it can use this "relating" element are very similar to the different possibilities for a question element in a WH- question, and in fact the two share part of the network. Here we use the letters PEL to indicate we are talking about a relative clause, so the feature PPEP* in Network 1 becomes PPEPREL. In sentences (s2) through (s22), we illustrated the different types of WH- questions. We can show parallel sentences for WHRS RSO clauses. The following list shows some examples and the relevant feature names:

- (s32) the thing with which you erased it PPEPREL
 (s33) the thing that you erased it with PPEPREL DANGLING
 (s34) the reason why the chicken crossed the road PELADJ
 (s35) the day when you were born PELADJ
 (s36) the way we will tell her the news PELADJ
 (s37) the place my little dog has gone PELADJ
 (s38) the reason why PELADJ SHORTPEL
- (s39) the hand which rocks the cradle SUBJPEL
 (s40) the number of Puerto Picans there are in Boston SUBJPEL

- (s41) the color her hair was last week COMPPEL
 (s42) the depth the ocean will be MEASPEL
 (s43) the information that you want OBJ1PEL
 (s44) the man you gave the book OBJ1PEL
 (s45) the book which you gave him OBJ2PEL
 (s46) the man to whom you gave the book TPANSTO2PEL
 (s47) the man you gave the book to TPANSTO2PEL
 (s48) the day the iceman came TIMEPEL
 (s49) the car your brother said he was expecting us to
tell Jane to buy DOWNPEL
 (s50) the state you said Lincoln was born in DOWNPEL

Notice that in sentences 36, 37, 40, 41, 42, 44, 47, 48, 49, and 50, there is no relative word like "which" or "that". These could just as well all have been put in, but English gives us the option of omitting them. When they are absent, the CLAUSE is marked with the feature PELDEL.

Returning to our network, we see that there is one other type of basic clause, the PSNG. This is a clause which is rank-shifted to serve as a NG. It can function as a part of another clause, a preposition group, or an adjective group. There are four basic types. The first two are TO and ING, as in:

- (s51) I like to fly. TO
 (s52) Building houses is hard work. ING
 (s53) He got it by saving coupons. ING

Notice that in s51, the PSNG clause is the object (OBJ1), in s52 it is the subject (SUBJ), and in s53 it is the object of a preposition (PPEPOBJ). We can have a separate subject within the TO and ING clauses, giving us the features SUBTO and SUBING:

- (s54) I wanted Ruth to lead the revolution. SUBTC
 (s55) They liked John's leading it. SUBING

The SUBING form takes its subject in the possessive.

In addition to ING and TC, we have the PEPOPT CLAUSE, which has the structure of an entire sentence, and is used as a participant in a relation about things like hearing, knowing, and saying:

- (s56) She heard that the other team had won.
 (s57) That she wasn't there surprised us.
 (s58) I knew he could do it.

The word "that" is used in s56 and s57 to mark the beginning of the PEPOPT CLAUSE, so they are assigned the feature THAT. The absence of "that" is left unmarked.

If the subject of a clause is in turn a PSNG clause, we may have trouble understanding it:

- (s59) That anyone who knew the combination could have opened
the lock was obvious.

There is a special mechanism for rearranging the sentence by using the word "it", so that the complicated subject comes last:

- (s60) It was obvious that anyone who knew the combination
could have opened the lock.

In this case, we say that the PSNG clause is serving as an ITSUBJ. TC and ING clauses can do the same:

- (s61) It will be fun to see them again.
 (s62) It was dangerous going up without a parachute.

The final type of PSNG is the VHRS, which is almost identical to the WHRS RSO described above. Rather than go through the details again, we will indicate how a few of our RSO examples (sentences s32 to s50) can be converted, and will leave the reader to do the rest.

- (s63) I don't know what he erased it with. PPEPPEL DANGLING
 (s64) Ask him when he was born. RELADJ
 (s65) He told me why. RELADJ SHOFTPEL
 (s66) It is amazing how many Puerto Ricans there are in Boston. SUPJTREL
 (s67) Only her hairdresser knows what color her hair was. COMPEL
 etc.

Let us examine one case more carefully:

- (s68) I knew which car your brother said that he was expecting us to tell Jane to buy.

Here we have a DOWNPEL clause, "which car....buy", serving as the object of the CLAUSE "I knew...". However, this means that somewhere below, there must be another clause with a slot into which the relative element can fit. In this case, it is the PSNG TO clause "Jane to buy", which is missing its object. This clause then has the feature UPREL, which indicates that its missing constituent is somewhere above in the structure. More specifically it is OBJIUPPEL.

In addition to the systems we have already described,

there is a TRANSITIVITY system for the CLAUSE, which describes the number and nature of its basic constituents. We mentioned earlier that a CLAUSE had such components as a subject and various objects. The transitivity system specifies these exactly. We have adopted a very surface-oriented notion of transitivity, in which we note the number and basic nature of the objects, but do not deal with their semantic roles, such as "range" or "beneficiary". Halliday's analysis (see reference (22)) is somewhat different, as it includes aspects which we prefer to handle as part of the semantic analysis. Our simplified network is:

A THERE CLAUSE has only a subject, marked SUBJT, while an INT CLAUSE has a SUBJECT and a COMPLEMENT. The COMPLEMENT can be either a NG, as in s70 or:

(s71) He was an agent of the FBI.

or a PREPG:

(s72) The king was in the counting house.

or an ADJG:

(s73) Her strength was fantastic.

(s74) My daddy is stronger than yours.

Other clauses are divided according to the number and type of objects they have. A CLAUSE with no objects is intransitive (ITRNS):

(s75) He is running.

With one object it is transitive (TRANS):

(s76) He runs a milling machine.

With two objects TRANS2:

(s77) I gave my love a cherry.

Some verbs are of a special type which use a location as a second object. One example is "put", as in:

(s78) Put the block on the table.

Note that this cannot be considered a TRANS with a modifier, as in:

(s79) He runs a milling machine in Chicago.

since the verb "put" demands that the location be given. We

cannot say "Put the block." This type of CLAUSE is called TRANSL, and the location object is the LOBJ. The LOBJ can be a PREPG as in s79, or a special adverb, such as "there" or "somewhere", as in:

- (s80) Where did you put it? or
 (s81) Put it there.

Some intransitive verbs also need a locational object for certain meanings, such as:

- (s82) The block is sitting on the table.

This is called ITPNSL.

Finally, there are INTensive clauses which are not BE clauses, but which have a COMPLEMENT, as in:

- (s83) He felt sick, and
 (s84) He made me sick.

We have not run into these with our simple subject matter, and a further analysis will be needed to handle them properly.

Any of the constituents we have been mentioning can be modified or deleted when these features interact with the features described in Network 1. For example in:

- (s85) the block which I told you to put on the table
 the underlined CLAUSE is TRANSL, but its OBJ1 is missing since it is an UPPEL.

English has a way of making up new words by combining a verb and a "particle" (PPT), producing a combination like

"pick up", "turn on", "set off", or "drop out". These do not simply combine the meanings of the verb and particle, but there is a special meaning attached to the pair, which may be very different from either word in isolation. Our dictionary contains a table of such pairs, and the grammar programs use them. A CLAUSE whose verb is a part of PPT pair has the feature PPT. The particle can appear either immediately after the word:

(s86) He threw away the plan.

or in a displaced position (marked by the feature DPPT):

(s87) He threw the plans away.

Regardless of whether there is a PPT or not, we have the choice between the features passive (PASV) and active (ACTV). ACTV places the semantic subject first:

(s88) The President started the war.

while PASV puts the semantic object first:

(s89) The war was started by the President.

If there is a PPEPG beginning with "by", it is interpreted as the semantic subject (as in s89), and the CLAUSE has the feature AGENT.

If the CLAUSE is active and its subject is a PSNG CLAUSE, we can use the IT form described earlier. This is marked by the feature IT, and its subject is marked ITSUBJ, as in sentences 60, 61, and 62.

2.3.4 Noun Groups

The best way to explain the syntax of the NOUN GROUP is to look at the "slot and filler" analysis, which describes the different components it can have. Some types of NG, such as those with pronouns and proper nouns, will not have this same construction, and they will be explained separately later.

We will diagram the typical NG structure, using a "*" to indicate that the same element can occur more than once. Most of these "slots" are optional, and may or may not be filled in any particular NG. The meanings of the different symbols are explained below.

```

-----
!   !   !   !   !   !
DET OPD NUM ADJ* CLASF* NOUN O*

```

NG Structure

The most important ingredient is the NOUN, which is almost always present (if it isn't, the NG is INCOMPLETE). It gives the basic information about the object or objects being referred to by the NG. Immediately preceding the NOUN, there are an arbitrary number of "classifiers" (CLASF). Examples of CLASF are:

- (s90) plant life
- (s91) the water meter cover adjustment screw

Notice that the same class of words can serve as CLASF and NOUN -- in fact Halliday uses one word class (called NOUN), and distinguishes between the functions of "head" and "classifier". We have separated the two because our dictionary gives the meaning of words according to their word class, and nouns often have a special meaning when used as a CLASF.

Preceding the CLASFs we have adjectives (ADJ), such as "big beautiful soft red..." We can distinguish adjectives from classifiers by the fact that adjectives can be used as the complement of a BE CLAUSE, but classifiers cannot. We can say "red hair", or "horse hair", or "That hair is red.", but we cannot say "That hair is horse.", since "horse" is a CLASF, not an ADJ. Adjectives can also take on the COMPARative and SUPERlative forms ("red, redder, and reddest"), while classifiers cannot ("horse, horser, and horsest"!!?).

Immediately following the NOUN we can have various qualifiers (Q), which can be a PREPG:

(s92) the man in the moon

or an ADJG:

(s93) a night darker than doom

or a CLAUSE PSO:

(s94) the woman who conducts the orchestra

We have already discussed the many types of PSO clauses. In later sections we will discuss the PPEPG and ADJG types which can occur as qualifiers.

Finally, the first few elements in the NG work together to give its logical description -- whether it refers to a single object, a class of objects, a group of objects, etc. The determiner (DET) is the normal start for a NG, and can be a word such as "a", or "that", or a possessive. It is followed by an "ordinal" (ORD), such as "first", "second", "last", or "only". There is an infinite sequence of number ordinals ("first, second, third...") and a few others such as "last" and "next". These can be recognized since they are the only words that can appear between a DET like "the" and a number, as in:

(s95) the next three days

Finally there is a NUMBER. It can either be a simple integer like "one", "two", etc. or a more complex construction such as "at least three", or "more than a thousand". It is possible for a NG to have all of its slots filled, as in:

DET	ORD	NUM	ADJ	ADJ	CLASF	CLASF	NOUN
the	first	three	old	red	city	fire	hydrants
		O(PPEPG)			O(CLAUSE)		
			without	covers	you	can	find

It is also possible to have combinations of almost any

TPRONG marks a NG whose head is a special TPPON, like "something", "everything", "anything", etc. These enter into a peculiar construction containing only the head and qualifiers, and in which an adjective can follow the head, as in:

(s95) anything green which is bigger than the moon

The feature PROPNG marks an NG made up of proper nouns, such as "John", "Oklahoma", or "The Union Of Soviet Socialist Republics."

These three special classes of NG do not have the structure described above. The PRONG is a single PRONoun, the PPOPNG is a string of PPOPNS, and the TPRONG has its own special syntax. The rest of the NGs are the unmarked (normal) type. They could be classified according to exactly which constituents are present, but in doing so we must be aware of our basic goals in systemic grammar. We could note whether or not a NG contained a CLASF or not, but this would be of minor significance. On the other hand, we do note, for example, whether it has a DET, and what type of DET it has, since this is of key importance in the meaning of the NG and the way it relates to other units. We distinguish between those with a determiner (marked DET) and those without one (NDET), as in:

(s97) Cats adore fish. NDET

TPRONG marks a NG whose head is a special TPPON, like "something", "everything", "anything", etc. These enter into a peculiar construction containing only the head and qualifiers, and in which an adjective can follow the head, as in:

(s95) anything green which is bigger than the moon

The feature PROPNG marks an NG made up of proper nouns, such as "John", "Oklahoma", or "The Union Of Soviet Socialist Republics."

These three special classes of NG do not have the structure described above. The PRONG is a single PRONoun, the PROPNG is a string of PPOPNS, and the TPPONG has its own special syntax. The rest of the NGs are the unmarked (normal) type. They could be classified according to exactly which constituents are present, but in doing so we must be aware of our basic goals in systemic grammar. We could note whether or not a NG contained a CLASF or not, but this would be of minor significance. On the other hand, we do note, for example, whether it has a DET, and what type of DET it has, since this is of key importance in the meaning of the NG and the way it relates to other units. We distinguish between those with a determiner (marked DET) and those without one (NDET), as in:

(s97) Cats adore fish. NDET

(s98) The cat adored a fish. DET

The DET can be DEFINite (like "the" or "that"), INDEFInite (like "a" or "an"), or a quantifier (ONTFP) (like "some", "every", or "no"). The DEFINite determiners can be either DEMonstrative ("this", "that", etc.) or the word "the" (the unmarked case), or a POSSESSive NG. The NG "the farmer's son" has the NG "the farmer" as its determiner, and has the feature POSES to indicate this.

An INDEF NG can have a number as a determiner, such as:

(s99) five gold rings
 (s100) at least a dozen eggs

in which case it has the feature NUMDET, or it can use an INDEF determiner, such as "a". In either case it has the choice of being a QUESTION. The question form of a NUMDET is "how many", while for other cases it is "which" or "what".

Finally, an NG can be determined by a quantifier (ONTFP). Although quantifiers could be subclassified along various lines, we do so in the semantics rather than the syntax. The only classifications used syntactically are between singular and plural (see below), and between NEGative and non-negative.

If a NG is either NUM'D or ONTFP, it can be of a special type marked OF, like:

- (s101) three of the offices
 (s102) all of your dreams

An OF NG has a DETerminer, followed by "of", followed by a DEFinite NG.

A determined NG can also choose to be INCOmplete, leaving out the NOUN, as an

- (s103) Give me three.
 (s104) I want none.

Notice that there is a correspondence between the cases which can take the feature OF, and those which can be INCOm. We cannot say either "the of them" or "Give me the." Possessives are an exception (we can say "Give me Juan's." but not "Juan's of them"), and are handled separately (see below).

The middle part of Network 3 describes the different possible functions a NG can serve. In describing the CLAUSE, we described the use of an NG as a SUPJ, COMP, and OBJECTS of various types. In addition, it can serve as the object of a PPEPG (PPEPOBJ), in:

- (s105) the rape of the lock

If it is the object of "of" in one of our special OF NGs, it is called an OFOBJ:

- (s106) none of your tricks

A NG can also be used to indicate TIME, as in:

- (s107) Yesterday the world ended.

(s108) The day she left, all work stopped.

Finally, a NG can be the POSSessive determiner for another NG. In:

(s109) the cook's kettles

the NG "the cook" has the feature POSS, indicating that it is the determiner for the NG "the cook's kettle", which has the feature POSES.

When a PRONG is used as a POSS, it must use a special possessive pronoun, like "my", "your", etc. We can use a POSS in an incomplete NG, like

(s110) Show me yours.

(s111) John's is covered with mud.

There is a special class of pronouns used in these NG's (labelled DEFPASS), such as "yours", "mine", etc.

Continuing to the last part of Network 3, we see features of person and number. These are used to match the noun to the verb (if the NG is the subject) and the determiner, to avoid ungrammatical combinations like "these kangaroo" or "the women wins". In the case of a PRONG, there are special pronouns for first, second, and third person, singular and plural. The feature NFS occurs only with the first-person singular pronouns ("I", "me", "my", "mine"), and no distinction is made between other persons, since they have no effect on the parsing. All singular

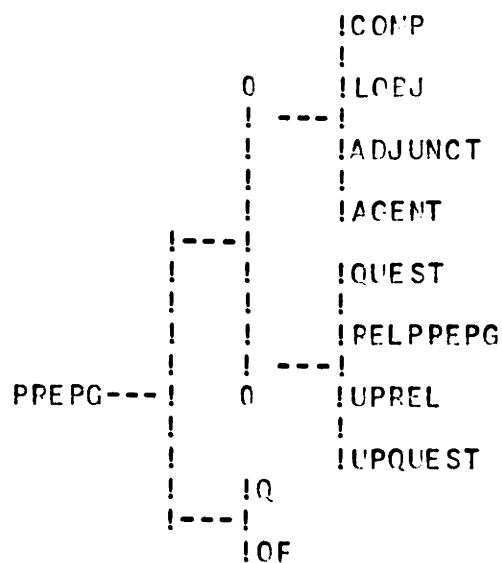
pronouns or other singular NGs are marked with the feature NS. The pronoun "you" is always treated as if it were plural and no distinction is made between "we", "you", "they", or any plural (NPL) NG as far as the grammar is concerned. Of course there is a semantic difference, which will be considered in later chapters.

2.3.5 Preposition Groups

The PPEPG is a comparatively simple structure used to express a relationship. It consists of a PPEPosition followed by an object (PPEPOBJ), which is either a NG or a PSNG CLAUSE. In some cases, the preposition consists of a two or three word combination instead of a single word, as in:

- (s112) next to the table
- (s113) on top of the house

The grammar includes provision for this, and the dictionary lists the possible combinations and their meanings. The words in such a combination are marked as PPEP2. The network for the PPEPG is:



NETWORK 4

The PREPG can serve as a constituent of a CLAUSE in several ways. It can be a COMPLEMENT:

(s114) Is it in the kitchen?

a locational object (LOBJ):

(s115) Put it on the table.

an ADJUNCT:

(s116) He got it by selling his soul.

or an AGENT:

(s117) It was bought by the devil.

If the PREPG is a constituent of a QUESTION CLAUSE, it can be the question element by having a QUESTION NG as its object:

(s118) in what city

(s119) for how many days

(s120) by whom

in which case the PREPG is also marked QUEST. A PREPPEL CLAUSE contains a PELPREPG:

(s121) the place in which she works

If the CLAUSE is an UPQUEST or an UPPEL, the PREPG can be the constituent which is "missing" the piece which provides the upward reference. In this case it is also marked UPREL:

(s122) the lady I saw you with
or UPQUEST:

(s123) Who did you knit it for?

In these cases, it is also marked SHOPT to indicate that the object is not explicitly in the PREPG. It can also be short if it is a PREPG in a DANGLING PREPO or PREPPEL CLAUSE:

(s124) what do you keep it in?

Within a NG, a PREPG serves as a qualifier (0):

(s125) the man in the iron mask
or as the body of an OF NG:

(s126) some of the people

2.3.6 Adjective Groups

The ADJG is a specialized unit serving as a COMPLEMENT of an intensive clause, as a Qualifier to an NG, or as a CLAUSE ADJUNCT. The network is:

```

!Q-----!      !THAN
!      !      !----!
!COMP-----!    !----!    !AS
!      !      !----!
!ADJUNCT---!    !COMPAR
!      !      !
!      !      !QUEST
!      !      !
!      !      !----

```

NETWORK 5

An ADJG which serves as an ADJUNCT contains an adverb, like "fast" in:

(s127) He could run faster than an arrow.

in place of an adjective. (Clearly our terminology could do with some cleaning up at places like this in doing a theoretical version of the grammar.) The other two types of ADJG use an adjective, as in a Qualifier:

(s128) a hotel as bad as the other one

or a Complement:

(s129) They were blissful.

The basic forms for an ADJG include THAN:

(s130) holier than thou

AS:

(s131) as quick as a flash

COMPARative:

(s132) This one is bigger.

or QUESTION:

(s133) How well can he take dictation?

The network is arranged to show that a qualifier ADJG can be only of the first two forms -- we cannot say "a man bigger" without using "than", or say "a man big". In the special case of a TPRON such as "anything" as in:

(s134) anything strange

the word "strange" is considered an ADJ which is a direct constituent of the NG, rather than an ADJG.

2.3.7 Verb Groups

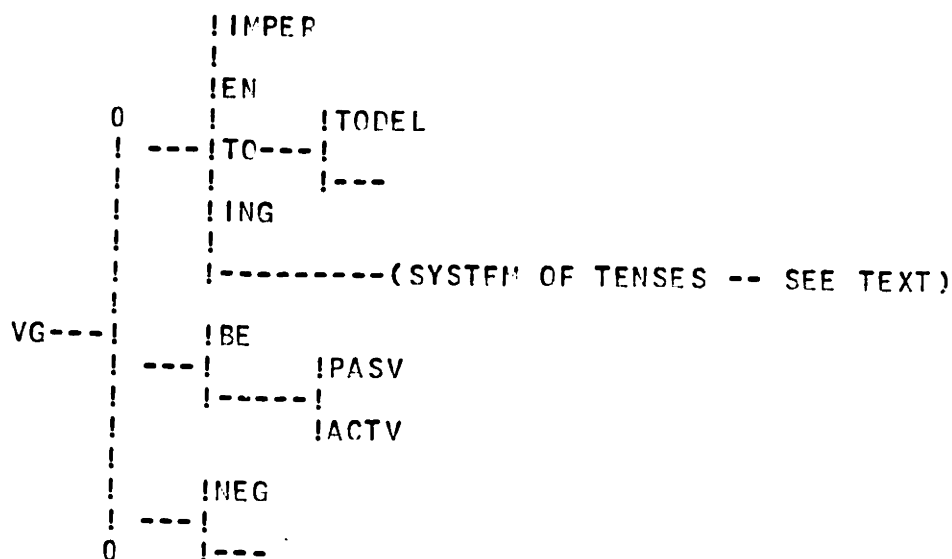
The English verb group is designed to convey a complex combination of tenses so that an event can relate several time references. For example, we might have:

(s135) By next week you will have been
living here
for a month.

This is said to have the tense "present in past in future". Its basic reference is to the future -- "next week", but it refers back to the past, and also indicates that the event is still going on at the time.

A VG can have a combination of "past", "present", "future", and "modal" which is restricted by a set of simple ordering relations. These are described in (Halliday (21)), and we will not give the details here. This tense system is not easily describable by a simple network -- in fact, the actual PROGRAMMAR program is a quite useful representation. In the network we simply refer to this system. The network

is:



NETWORK 6

We have several types of VG which do not enter the normal tense system, but which have a specialized form. The IMPER VG is used in imperatives:

- (s136) Fire when ready.
 (s137) Don't drop the baby.

It consists of a verb in the INFinitive form, possibly preceded by the auxilliary "do" or its negative form "don't". The EN VG is used in EN RSO CLAUSES, like:

- (s138) a man forsaken by his friends

and consists of a past participle verb. The ING VG is made up of an ING verb or the verb "being" followed by an EN verb. It is used in various types of ING clauses:

- (s139) Being married is great.

(s140) the girl sitting near the wall

Similarly, the TO VG is used in TO clauses. In the case of conjoined structures, the "to" may be omitted from the second clause, as in:

(s141) We wanted to stop the war and end repression.
Such a VG is marked TODEL.

We separate those verb groups whose main verb is "be" from the others, as they do not undergo the further choice between PASV and ACTV. These correspond to the same features for clauses, and are seen in the structure by the fact that a PASV VG contains a form of the auxiliary "be" followed by the main verb in the EN form, as in:

(s142) The paper was finished by the deadline.
(s143) He wanted to be kissed by the bride.

Finally, any VG can be NEGative, either by using a negative form of an auxiliary like "don't", "hasn't", or "won't", or by including the word "not".

2.3.8 Words

In our grammar we have a number of separate word classes, each subdivided into subclasses by the features they have. It was necessary to make arbitrary decisions as to whether a distinction between groups of words should be considered as different classes or different features within the same class. Actually we could have a much more tree-

like structure of word classes, in which the ideas of classes and features were combined. Since we have not yet done this, we will present a list of the different classes in alphabetical order, and for each of them give descriptions of the relevant features. Many words can be used in more than one class, and some classes overlap to a large degree (such as NOUN and CLASF). In our dictionary, we simply list all of the syntactic features the word has for all of the classes to which it can belong. When the parser parses a word as a member of a certain class, it sorts out those features which are applicable.

ADJ -- Adjective is one of the constituents of a NG as well as being the main part of an ADJG. This class includes words like "big", "ready", and "strange". The only features are SUPERlative (as in "biggest") and COMPARative (as in "bigger").

ADV -- We use the name "adverb" to refer to a whole group of words used to modify other words or clauses. It is sort of a "mixed bag" of things which don't really fit anywhere else. The basic classification depends on what is being modified, and has the terms (ADVADV VPAD PREPADV CLAUSEADV). An ADVADV is a word like "very" which modifies other adverbs and adjectives. A VPAD modifies verbs, and includes the class of words ending in "-ly"

like "quickly" and "easily". A PPREADV modifies prepositions, as "directly" in "directly above the stove". A CLAUSEADV is a constituent of a clause, and can be either TIMW or PLACE. A TIMW like "usually", "never", "then", or "often" appears as a CLAUSE constituent specifying the time. The PLACE ADV "there" can either be an adjunct, as in:

(s144) There I saw a miracle.

or an LOBJ, as in:

(s145) Put it there.

BINDER -- Binders are used to "bind" a secondary clause to a major clause, as in:

(s146) Before you got there, we left.

(s147) I'll go if you do.

We do not assign any other features to binders.

CLASF -- In Section 2.3.3 we discussed the use of CLASF as a constituent of a NG. The CLASF is often another NOUN, but it appears in a position like an adjective, as in "boy scout".

DET -- DETerminers are used as a constituent of a NG, as described in 2.3.3. They can have a number of different features, as described in the network:

Not all quantifiers are QFD -- we cannot say "every of the cats" or "Buy every." Quantifiers can also be NEGative, like "none" or "no", or can be NONUM, indicating that they cannot be used with a number, such as "many" or "none" (we can say "any three cats" or "no three cats", but not "none three" or "many three"). The NG program takes these feature into account in deciding what NG constituents to look for. It also has to find agreement in number between the DET and the NOUN. A DET can have the features "singular" (NS), "plural" (NPL) or MASS (like "some" or "no", which can go with MASS nouns like "water"). A DET can have more than one of these -- "the" has all three, while "all" is MASS and NPL, and "a" is just NS.

NOUN -- The main constituent of a NG is its NOUN. It has a feature of number, identical to that of the DETerminers it must match. The word "parsnip" is NS, "parsnips" is NPL, and "wheat" is MASS. Some nouns may have more than one of these, such as "fish", which is all three since it can be used in "a fish", "three fish", or "fish is my favorite food." In addition, a NOUN can be POSSESSive, like "parsnip's".

In order to tell whether a NG is functioning as a time element in a CLAUSE, we need to know whether its

NCUN can refer to time. We therefore have two features -
- TIME words like "day", and "month", as in:

(s150) The next day it started to snow.

and TIM1 words like "yesterday" and "tomorrow".

NUM' -- The class of NUMbers is large (uncountably infinite) but not very interesting syntactically. For our purposes we only note the features NS (for "one") and NPL (for all the rest). In fact, our system does not accept numbers in numeric form, and has only been taught to count to ten.

NUMD -- In complex number specifications, like "at least three" or "more than a million", we have a NUMD. The features they can have are (NUMDAN NUMDAS NUMDAT NUMDALONE). NUMDAN words such as "more" and "fewer" are used with "than", while NUMDAS words such as "few" fit into the frame "as...as", and NUMDATs are preceded by "at", as in "at least", and "at most". NUMDALONE indicates that the NUMD can stand alone with the number, and includes "exactly" and "approximately".

CRD -- The class of ORDinals includes the ordinal numbers "first", "second", etc., and a few other words which can fit into the position between a determiner and a number, like "next", "last", and "only". Notice that SUPERlative ADJectives can also fill this slot in the NG.

PPEP -- Every PREPG begins with a PPEPosition, either alone, or as part of a combination such as "on top of". In the combination case, the words following the initial PPEP have the feature PPEP2. A PREP which cannot appear without a PPEP2 (such as "next" which appears in "next to") are marked NEED2.

PRON -- PRONouns can be classified along a number of dimensions, and we can think of a large multi-dimensional table with most of its positions filled. They have number features (NS NPL NFS) (note that instead of the more usual division into first, second, and third person, singular and plural, we have used a reduced one in which classes with the same syntactic behavior are lumped together). They can be POSSESSIVE, such as "your" or "my", or POSSEDEF, like "yours" or "mine". Some of the personal pronouns distinguish between a SUBJECT form like "I" and an OBJECT form like "me". We also have special classes like DEMonstrative ("this" and "that") and PRONPEL -- the pronouns used in relative clauses, such as "who", "which", and "that". Those which can be used as a question element, such as "which" and "who" are marked QUEST.

PPCPN -- Proper nouns include single words like "Carol", or phrases such as "The American Legion" which could be

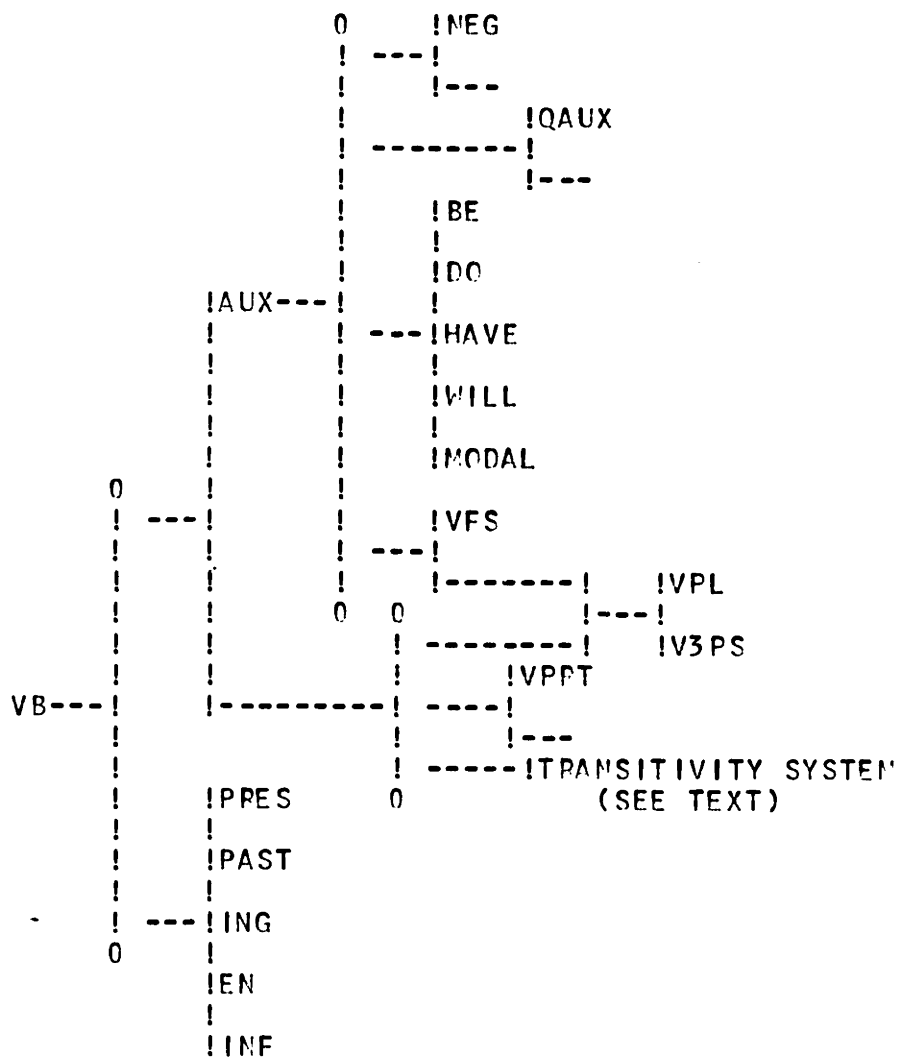
parsed, but are interpreted as representing a particular object (physical or abstract). A PROPN can be MPL or NS, and is assumed to be NS unless defined otherwise.

PPT -- In Section 2.3.3, we discussed clauses which use a combination of a "particle" and a verb, like "pick up" or "knock out". The second word of these is a PPT.

QADJ -- One class of QUESTION CLAUSE uses a QADJ such as "where", "when", or "how" as its question element. They can also be used in various kinds of relative clauses, as explained in Section 2.3.3.

TPRON -- There is a small class of words made up of a quantifier and the suffix "-thing" which enter into a special type of NG construction like "anything green". We cannot consider this an abbreviation for a quantifier followed by a noun, since the NG "any block green" would have the same structure but is not grammatical.

VB -- The verb has the most complex network of features of any word in our grammar. They describe its tense, transitivity, number, and use, as well as marking special verbs like "be". The network is:



NETWORK 8

We first divide verbs into AUXiliaries and others (unmarked). AUXiliaries are the "helping verbs" which combine with others in complex VG structures. They can have special NEGative forms, like "can't", or can appear standing alone at the beginning of a QUESTION, in which

case they have the function OAUX, as in:

(s151) Will I ever finish?

The auxiliaries include "be", "do", "have", "will", and the MODALS like "could" "can", and "must". We use separate features for these as they are critical in determining the structure of a VG. An AUX can choose from the system of person and number, distinguishing "third-person singular" (V3PS) as in "is", "plural", as in "have", or "first singular" (VFS), used only for "am".

Non-auxiliary verbs can be VPRT, which combine with a PPT, and they have a whole cluster of transitivity features. In Section 2.3.3 we described the different transitivity features of the CLAUSE, and these are controlled by the verb. We therefore have the features (TRANS ITPNS TRANS2 TRANSL ITRNSL INT) In addition, the verb can control what types of PSNG CLAUSE can serve as its various objects. The feature names combine the type of CLAUSE (ING TO REPORT SUBTO SUBING) with either -OB or -OB2, to get a product set of features like SUBTOB and INGOB2.

Finally, all of these kinds of verbs can be in various forms such as ING ("breaking"), EN ("broken"), INFinite ("break), PAST ("broke"), and PPEsent ("breaks"). The network does not illustrate all of the

relations, as some types (like MODAL) do not make all of these choices.

2.3.9 Following the Parser in Operation

Let us follow the parser through two examples to see how the grammar is used in practice. We will not actually watch all of the details, or deal with the way semantic programs are intermixed with the grammar. Instead we will follow a somewhat reduced version, to get a feeling for the way the grammar works, and the way it interacts with the different features described above. We have chosen one very simple sentence, and another which is difficult enough to exercise some of the more complex features of the grammar. The first sentence is the first sentence of our sample dialog (Section 1.3):

Pick up a big red block.

The system begins trying to parse a sentence, which, as explained above, means looking for a MAJOR CLAUSE. It therefore activates the grammar by calling (PARSE CLAUSE MAJOR). Since CLAUSE is one of our units, there is a program defined for it. The CLAUSE program is called with an initial feature list of (CLAUSE MAJOR).

The CLAUSE program looks at the first word, in order to decide what unit the CLAUSE begins with. If it sees an adverb, it assumes the sentence begins with a single-word

modifier. If it sees a PREPosition, it looks for an initial PREPG. If it sees a BINDER, it calls the CLAUSE program to look for a BOUND CLAUSE. In English (and possibly all languages) the first word of a construction often gives a very good clue as to what that construction will be. We have "advance notice" of what structures to look for, and this makes parsing much easier. Our grammar tries to make as much use as possible of these natural "signals". In this case, the initial word is a verb, and this indicates that we may have an IMPERATIVE CLAUSE. The program calls (PARSE VG IMPER) to start the VG program looking for a VG of the right type. The call sets up the initial VG feature list (VG IMPER). The VG program checks this initial feature list, and sees that it is looking for an IMPERative VG. This must either begin with some form of the verb "do", or with the main verb itself. Since the next word is not "do", it calls (PARSE VB INF (MVB)). This is a different kind of call to PARSE, since VB is not a unit we have defined. It is a word class, and the call says to check the next word in the input (in this case still the first word) to see whether it is indeed the INFinitive form of a VerB. If so, it is to be attached to the parsing tree, and given the additional feature MVB (main verb).

We use several conventions for diagramming syntactic

structures. Rather than using a tree format (which quickly grows off of the page), we use a format more like a traditional outline, with the sub-phrases of any phrase indicated immediately below it and indented. We use the symbol "*" to indicate the program which is currently active, and show the actual words of each piece of the sentence to the right of the outline, putting units larger than WOPD in parentheses. A series of dots in the English words to the right of a unit indicates that the program for that unit has not yet finished. We can diagram our current structure:

```
(CLAUSE MAJOR)                (pick...)
  (VG IMPER)*                  (pick...)
    (VB MVB INF TRANS VPRT)   pick
```

This says that we have a CLAUSE, with a constituent which is a VG, and that the VG program is active. The VG so far consists of only a VB. Notice that some new properties have appeared on the list for VB. We have not mentioned TRANS or VPRT. These came from the definition of the word "pick" when we called the function PAPSE for a word (see section 2.4.3 for details).

Ordinarily the VG program checks for various kinds of tense and number, but in the special case of an IMPER VP, it returns immediately after finding the verb. We will see other cases in the next example.

When the VG program succeeds, CLAUSE takes over again. Since it has found the right kind of VG for an IMPERATIVE CLAUSE, it puts the feature IMPEP on the CLAUSE feature list. It then checks to see whether the MVB has the feature VPRT, indicating it is a special kind of verb which takes a particle. It discovers that "pick" is such a verb, and next checks to see if the next word is a PRT, which it is. It then checks in the dictionary to see if the combination "pick up" is defined, and when it discovers this is true, it calls (PARSE PRT) to add "up" to the parsing tree. Notice that we might have let the VG program do the work of looking for a PRT, but it would have run into difficulties with sentences like "Pick the red block up." in which the PRT is displaced. By letting the CLAUSE program do the looking, the problem is simplified.

As soon as it has parsed the PRT, the CLAUSE program marks the feature PRT on its own feature list. It then looks at the dictionary entry for "pick up" to see what transitivity features are there. It is TRANSITIVE, which indicates that we should look for one object -- OBJ1. The dictionary entry does not indicate that this is a verb which can take special types of PSNG clauses as objects, so the object must be either a NG or a WHPS clause (which can appear wherever a NG can). If the object were a WHRS

clause, it would begin with a relative pronoun, like "Pick up what I told you to." Since our next word is "a", this is not the case, so the CLAUSE program looks for an object by calling (PARSE NG OBJ OBJ1), asking the NG program to find a NG which can serve as an OBJ1. Our structure is now:

(CLAUSE MAJOR IMPER PRT)	(pick up...)
(VG IMPER)	(pick)
(VB MVB INF TRANS PRT)	pick
(PRT)	up
(NG OBJ OBJ1)*	(...)

The NG program is started and notices that the upcoming word is a DET, "a". It calls (PARSE DET) to add it to the parsing tree, then uses the function TPNSF to transfer relevant features from the DET to the entire NG. It is interested in the type of determination (DEF vs. INDEF vs. QNTFP), and the number (NS vs. MPL). It also adds the feature DET to the NG to indicate that it has a determiner. The feature list for the NG is now:

(NG OBJ OBJ1 DET INDEF NS)

since "a" is a singular indefinite determiner. The NG program then notices the feature INDEF, and decides not look for a number or an ordinal (we can't say "a next three blocks"), or for the OF construction ("a of them" is impossible). It goes on immediately to look for an ADjective by calling (PARSE ADJ). When this succeeds with the next word "big", a simple program loop returns to the

(PARSE ADJ) statement, which succeeds again with "red". on the next trip it fails, and sends the program on to look for a classifier, since "block" isn't an ADJ. But "block" isn't a CLASF either in our dictionary, so the NG program goes on to look for a NOUN, by calling (PARSE NOUN). This succeeds with the NOUN "block", which is singular, and the program next checks to see if it agrees with the number features already present from the determiner (to eliminate illegal combinations like "these boy"). In this case, both are singular (NS), so the program is satisfied. Ordinarily it would go on to look for qualifiers, but in this case there is nothing left in the sentence. Remember that we have an especially easy way of indicating in a PROGRAMMAP program what action should be taken at any point if the sentence runs out. We can do it by simply putting a third tag in any branch statement. In this case, since we have found all of the basic constituents we need for a NG, the "third branch" tells us that the NG program should return success. If we had run out after the determiner, it would have sent us to check for an INCOMPLETE NG, while if we had run out after an ADJ it would have entered a backup program which would check to see whether we had misinterpreted a NOUN as an ADJ.

In this case, the NG program returns, and the CLAUSE program similarly notices that the sentence has ended.

Since a TRANS verb needs only one object, and that object has been found, the CLAUSE program marks the feature TRANS, and returns, ending the parsing. In actual use, a semantic program would be called here to understand and execute the command -- in fact, semantic programs would have been called at various points throughout the process. The final result looks like:

(CLAUSE MAJOR IMPER PPT TRANS) (pick up a big red block)

(VG IMPER)	(pick)
(VB MVB INF TRANS VPRT)	pick
(PRT)	up
(NG OBJ OBJ1 DET INDEF NS)	(a big red block)
(DET INDEF NS)	a
(ADJ)	big
(ADJ)	red
(NOUN NS)	block.

Now let us take a more complex sentence, like:

How many blocks are supported by the cube which I wanted you to pick up?

We will not go into as much detail, but will emphasize the new features exhibited by this example. First, the parser recognizes that this sentence is a question by its punctuation -- it ends with a question mark. This "cheating" is not really necessary, and in the future the grammar will be revised to look for the other signals of a question (for example, beginning with a determiner like "how many" or "which").

In any event, the feature QUESTION is noted, and the program must decide what type of question it is. It checks to see if the CLAUSE begins with a QADJ like "why", "where", etc. or with a PREPosition which might begin a PPEPG QUEST (like "In what year...").

All of these things fail in our example, so it decides the CLAUSE must have a NG as its question element, (called NGQ), marks this feature, and calls (PARSE NG QUEST). The NG program starts out by noticing QUEST on its initial feature list, and looking for a question determiner (DET QDET). Since there are only three of these ("which", "what", and "how many"), the program checks for them explicitly, parsing "how" as a QDET, and then calling (PARSE NIL MANY), to add the word "many" to the parsing tree, without worrying about its features. (The call (PARSE NIL X) checks to see if the next word is actually the word "x").

Since a determiner has been found, its properties are added to the NG feature list, (in this case, (NUMDET INDEF NPL)), and the NG program goes on with its normal business, looking for adjectives, classifiers, and a noun. It finds only the NOUN "pyramids" with the features (NOUN NPL). The word "pyramid" appears in the dictionary with the feature NS, but the input program which recognized the plural ending changed NS to NPL for the form "pyramids". Agreement is

checked between the NOUN and the rest of the NG, and since "how many" added the feature NPL, all is well. This time, there is more of the sentence left, so the NG program continues, looking for a qualifier. It checks to see if the next word is a PPEPosition (as in "pyramids on the table), a relative word ("pyramids which...), a past participle ("pyramids supported by...), an ING verb ("pyramids sitting on...) a comparative adjective ("pyramids bigger than...) or the word "as" ("pyramids as big as...). If any of these are true, it tries to parse the appropriate qualifying phrase. If not, it tries to find an PSO CLAUSE ("pyramids the block supports). In this case, all of these fail since the next word is "are", so the NG program decides it will find no qualifiers, and returns what it already has. This gives us:

(CLAUSE MAJOR QUESTION NGQ)*	(how many blocks...)
(NQ QUEST DET NUMDET NPL INDEF)	(how many blocks)
(DET ODET NPL INDEF)	how
()	many
(NOUN NPL)	blocks

Next the CLAUSE program wants a VG, so it calls (PARSE VG NAUX). The feature NAUX indicates that we want a VG which does not consist of only an AUXilliary verb, like "be" or "have". If we saw such a VG, it would indicate a structure like "How many pyramids are the boxes supporting?", in which the question NG is the object of the

CLAUSE. We are interested in first checking for the case where the question NG is the subject of the CLAUSE.

The VG program is designed to deal with combinations of auxiliary verbs like "had been going to be..." and notes that the first verb is a form of "be". It calls (PAPSE VB AUX BE), assuming that "are" is an auxiliary rather than the main verb of the sentence (if this turns out wrong, there is backup). It transfers the initial tense and person features from this verb to the entire VG (The English VG always uses the leading verb for these features, as in "He has been...", where it is "has" which agrees with "he") In this case "are" is plural (VPL) and present tense (PRES).

When "be" is used as an auxiliary, it is followed by a verb in either the ING or the EN form. Since "supported" is an EN form (and was marked that way by the input program), The VG program calls (PAPSE VB EN (MVB)), marking "supported" as the main verb of the clause. The use of a "be" followed by an EN form indicates a PASV VG, so the feature PASV is marked, and the VG program is ready to check agreement. Notice that so far we haven't found a SUBJECT for this clause, since the QUESTION NG might have been an object, as in "How many pyramids does the box support?" However the VG program is aware of this, and realizes that instead of checking agreement with the constituent marked

SUBJ, it must use the one marked QUEST. It uses PROGRAMMER's pointer-moving functions to find this constituent, and notes that it is NPL, which agrees with VPL. VG therefore is happy and returns its value. We now have:

(CLAUSE MAJOR QUESTION NGO)*	(how many blocks are supported...)
(NG QUEST DET NUMDET NPL INDEF)	(how many blocks)
(DET ODET NPL INDEF)	how
()	many
(NOUN NPL)	blocks
(VG NAUX VPL PASV (PPES))	(are supported)
(VB AUX BE PRES VPL)	are
(VB MVB EN TRANS)	supported

The CLAUSE program resumes, and marks the feature SUBJQ, since it found the right kind of VG to indicate that the NG "how many blocks" is indeed the subject. It next checks to see if we have a PRT situation as we did in our first example. We don't, so it next checks to see if the VG is PASV, and marks the clause with the feature PASV. This indicates that there will be no objects, but there might be an AGENT phrase. It checks that the next word is "by", and calls (PARSE PREPG AGENT).

The PREPG program is fairly simple -- it first calls (PARSE PPEP), then (PARSE NG OBJ PPEPOBJ). The word "by" is a PPEP, so the first call succeeds and NG is called and operates as described before, finding the DET "the" and the

NOUN "cube", and checking the appropriate number features. In this case, "the" is both NPL and NS, while "cube" is only NS, so after checking the NG has only the feature NS.

The NG program next looks for qualifiers, as described above, and this time it succeeds. The word "which" signals the presence of a RSQ WHRS CLAUSE modifying "cube". The NG program therefore calls (PARSE CLAUSE PSO WHRS). The parsing tree now looks like:

(CLAUSE MAJOP QUESTION NGQ SUBJO PASV)	(how many blocks are supported by the cube...)
(NG QUEST DET NUMDET NPL INDEF)	(how many blocks)
(DET QDET NPL INDEF)	how
()	many
(NOUN NPL)	blocks
(VG NAUX VPL PASV (PRES))	(are supported)
(VB AUX BE PRES VPL)	are
(VB MVB EN TRANS)	supported
(PREPG AGENT)	(by the cube...)
(PREP)	by
(NG OBJ PREPOBJ DET DEF NS)	(the cube...)
(DET DEF NPL NS)	the
(NOUN NS)	cube
(CLAUSE PSO WHRS)*	(...)

The CLAUSE program is immediately dispatched by the feature WHRS to look for a PELWD. It finds "which", and marks itself as NGREL. It then goes on to look for a (VG NAUX) just as our QUESTION NGQ clause did above. Remember that WH- questions and WHRS clauses share a great deal of

the network, and they share much of the program as well. This time the VG program fails, since the next word is "I", so the CLAUSE program decides that the clause "which I..." is not a SUBJREL. It adds the temporary feature NSUBREL, indicating this negative knowledge, but not deciding yet just what we do have. It then goes to the point in the normal clause program which starts looking for the major constituents of the clause -- subject, verb, etc. We call (PARSE NG SUBJ) and succeed with the PRONG "I". We then look for a VG, and find "wanted". In this case, since the verb is PAST tense, it doesn't need to agree with the subject (only the tenses beginning with PPES show agreement). The feature MAGR marks the non-applicability of agreement. The parsing tree from the WHPS node on down is now:

```
(CLAUSE PSQ WHRS NGPEL NSUBREL)*      (which I wanted...)
      (RELWD)                            which
      (NG SUBJ PRONG NFS)                 (I)
      (PPON NFS)                          I
      (VG NAGR (PAST))                    (wanted)
      (VB MVB PAST TRANS TOOBJ SUBTOBJ)  wanted
```

The CLAUSE program notes that the MVB is TRANS and begins to look for an OBJ1. This time it also notes that the verb is a TOOBJ and a SUBTOBJ (it can take a TC clause as an object, as in "I wanted to go", or a SUBTC, as in "I

wanted you to go." Since the next word isn't "to", it decides to look for a SUBTO clause, calling (PARSE CLAUSE RSNG OBJ OBJ1 SUBTO). In fact, this checking for different kinds of RSNG clauses is done by a small function named PARSEREL, which looks at the features of the NVE, and calls the appropriate clauses. PARSEREL is used at several points in the grammar, and one of main advantages of writing grammars as programs is that we can write such auxiliary programs (whether in PROGMAP or LISP) to make full use of regularities in the syntax.

The CLAUSE program is called recursively to look for the SUBTO clause "you to pick up". It finds the subject "you", and calls (PARSE VG TO) since it needs a verb group of the "to" type. The VG program notices this feature and finds the appropriate VG (which is again NAGP). The PRT mechanism operates as described in the first example, and the bottom of our structure now looks like:

(CLAUSE PSQ WHPS NGPEL NSUBPEL)	(which I wanted you to pick up)
(PELVD)	which
(NG SUBJ PRONG NFS)	(I)
(PRON NFS)	I
(VG NAGR (PAST))	(wanted)
(VB MVB PAST TRANS TOOBJ SUBTOBJ)	wanted
(CLAUSE PSNG SUPTO OBJ OBJ1 PRT)*	(you to pick up)
(NG SUBJ PRONG NPL)	(you)
(PPON NPL)	you
(VG TO NAGP)	(to pick)
()	to
(VB MVB INF TRANS VPRT)	pick
(PRT)	up

Notice that we have a transitive verb-particle combination, "pick up", with no object, and no words left in the sentence. Ordinarily this would cause the program to start backtracking -- checking to see if the MVB is also intransitive, or if there is some way to reparse the clause. However we are in the special circumstance of an embedded clause which is somewhere on the parsing tree below a relative clause with an "unattached" relative. In the clause "which I told you to pick up", I is the subject, and the CLAUSE "you to pick up" is the object. The "which" has not been related to anything. There is a small program named UPCHECK which uses PROGPARMAP's ability to look around on the parsing tree. It looks for this special situation, and when it finds it does three things: 1) Mark the current clause as UPPEL, and the appropriate type of UPPEL

for the thing it is missing (in this case OBJ1UPPEL). 2) Remove the feature NSUBPEL from the clause with the unattached relative 3) Replace it with DOWNPEL to indicate that the relative has been found below. This can all be done with simple programs using the basic PROGRAMMAP primitives for moving around the tree (see section 2.4.8) and manipulating features at nodes (see 2.4.9). The information which is left in the parsing tree is sufficient for the semantic routines to figure out the exact relationships between the various pieces involved.

In this example, once the CLAUSE "to pick up" has been marked as OBJ1UPPEL, it has enough objects, and can return success since the end of the sentence has arrived. The CLAUSE "which I want you to pick up" has an object, and has its relative pronoun matched to something, so it also succeeds, as does the PG "the cube...", the PREPG "by the cube..", and the MAJOR CLAUSE. The final result is:

(CLAUSE MAJOR QUESTION NGQ SUBJO PASV AGENT)

(NG QUEST DET NUMDET NPL INDEF)	(how many blocks)
(DET QDET NPL INDEF)	how
()	many
(NOUN NPL)	blocks
(VG NAUX VPL PASV (PRES))	(are supported)
(VB AUX BE PRES VPL)	are
(VB MVB EN TRANS)	supported
(PREPG AGENT)	(by the cube which I wanted you to pick up)
(PREP)	by
(NG OBJ PREPOBJ DET DEF NS)	(the cube which I wanted you to pick up)
(DET DEF NPL NS)	the
(NOUN NS)	cube
(CLAUSE RSO WHRS NGREL DOWNREL TRANS)	(which I wanted you to pick up)
(PELWD)	which
(NG SUBJ PRONG NFS)	(I)
(PRON NFS)	I
(VG NAGR (PAST))	(wanted)
(VB MVB PAST TRANS TOOBJ SUBTOBJ)	wanted
(CLAUSE PSNG SUBTO OBJ OBJ1 PPT TRANS	(you to pick up)
UPREL OBJ1UPREL)	
(NG SUBJ PRONG NPL)	(you)
(PRON NPL)	you
(VG TO NAGR)	(to pick)
()	to
(VB MVB INF TRANS VPPT)	pick
(PPT)	up

Even in this fairly lengthy description, we have left out much of what was going on. For example we have not mentioned all of the places where the CLAUSE program checked for adverbs (like "usually" or "quickly"), or the VG program

looked for "not", etc. These are all "quick" checks, since there is a PROGRAMMAR command which checks the features of the next word. In following the actual programs, the course of the process would be exactly as described, without backups or other attempts to parse major structures.

This may seem like a quite complex process and complex grammar, compared to other systems, or even our own examples in Section 2.2. This is because language is indeed a highly complex phenomenon. We have tried to handle a great deal more of the complexity of English than any of the previous language-understanding systems. It is only due to the fact that PROGRAMMAR gives us an easy framework in which to include complexity that it was at all possible to include such a detailed grammar as only one part of a project carried out by a single person in less than two years.

looked for "not", etc. These are all "quick" checks, since there is a PROGPARMAP command which checks the features of the next word. In following the actual programs, the course of the process would be exactly as described, without backups or other attempts to parse major structures.

This may seem like a quite complex process and complex grammar, compared to other systems, or even our own examples in Section 2.2. This is because language is indeed a highly complex phenomenon. We have tried to handle a great deal more of the complexity of English than any of the previous language-understanding systems. It is only due to the fact that PROGRAMMAP gives us an easy framework in which to include complexity that it was at all possible to include such a detailed grammar as only one part of a project carried out by a single person in less than two years.

2.4 Programming Details

2.4.1 Operation of the System

Since the grammar is itself a program, there is not much overhead mechanism needed for the basic operation of the parser. Instead, the system consists mostly of special functions to be used by the grammar. The system maintains a number of global variables, and keeps track of the parsing tree as it is built by the main function, PARSE. When the function PARSE is called for a UNIT which has been defined as a PROGRAMMAP program, the system collects information about the currently active node, and saves it on a pushdown list. It then sets up the necessary variables to establish a new active node, and passes control to the PROGRAMMAP program for the appropriate unit. If this program succeeds, the system attaches the new node to the tree, and returns control to the node on the top of the PDL. If it fails, it restores the tree to its state before the program was called, then returns control. A PROGRAMMAP program is actually converted by a simple compiler to a LISP program and run in that form. The variables and functions available for writing PROGRAMMAP programs are described in the rest of part 2.4. Sections 2.4.1 to 2.4.5 explain special features of the language. Sections 2.4.6 to 2.4.9 are more in the style of a manual which would allow the reader to understand

PROGRAMMAP programs. In order to make these details more independent of our detailed grammar of English, we will continue to use a simplified grammar whenever possible. We use the hypothetical grammar begun in 2.2, and try to use full length feature names for easier understanding.

When the function PARSE is called with a first argument which has not been defined as a PROGRAMMAP program, it checks to see whether the next word has all of the features listed in the arguments. If so, it forms a new node pointing to that word, with a list of features which is the intersection of the list of features for that word with the allowable features for the word class indicated by the first argument of the call. For example, the word "blocks" will have the possibility of being either a plural noun or a third-person-singular present-tense verb. Therefore, before any parsing it will have the features (NOUN VEPP N-PL VP-3PS TRANSITIVE PPRESENT). If the expression (PARSE VEPP TRANSITIVE) is evaluated when "blocks" is the next word in the sentence to be parsed, the feature list of the resulting node will be the intersection of this combined list with the list of allowable features for the word-class VEPP. If we have defined:

```
(DEFPOP VERB (VEPP INTPANSITIVE TRANSITIVE PPRESENT
PAST VB-3PS VB-PL) ELIM),
```

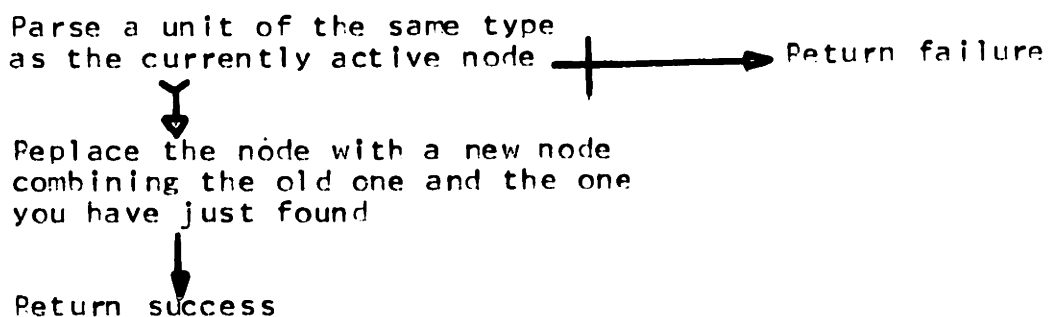
the new feature list will be (VEPB TRANSITIVE PPRESNT VB-3PS). (ELIM is simply a property indicator chosen to indicate this list which ELIMinates features). Thus, even though words may have more than one part of speech, when they appear in the parsing tree, they will exhibit only those features relevant to their actual use in the sentence.

2.4.2 Special Words

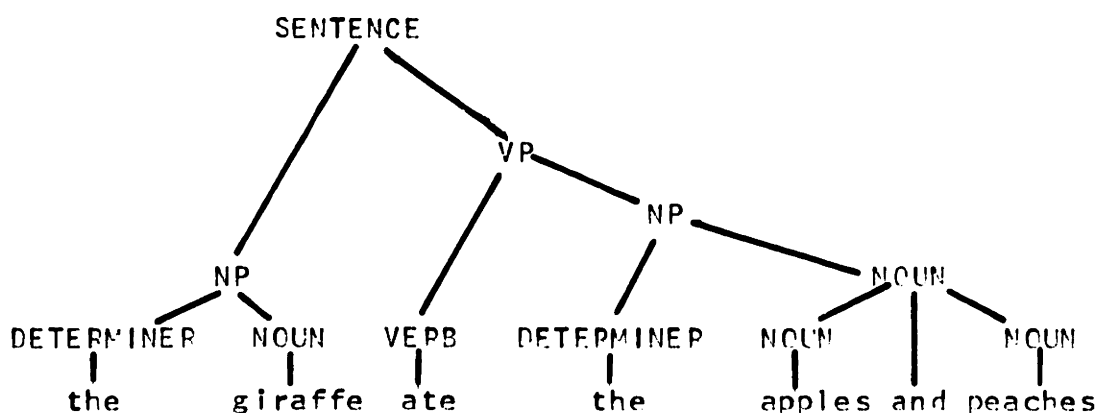
Some words must be handled in a very special way in the grammar. The most prevalent are conjunctions, such as "and" and "but". When one of these is encountered, a program should be called to decide what steps should be taken in the parsing. This is done by giving these words the grammatical features SPEC or SPECL. Whenever the function PARSE is evaluated, before returning it checks the next word in the sentence to see if it has the feature SPEC. If so, the SPEC property on the property list of that word indicates a function to be evaluated before parsing continues. This program can in turn call PROGRAMMAP programs and make an arbitrary number of changes to the parsing tree before returning control to the normal parsing procedure. SPECL has the same effect, but is checked for when the function PARSE is called, rather than before it returns. Various other special variables and functions allow these programs to control the course of the parsing process after they have

been evaluated. By using these special words, it is possible to write amazingly simple and efficient programs for some of the aspects of grammar which cause the greatest difficulty. This is possible because the general form of the grammar is a program.

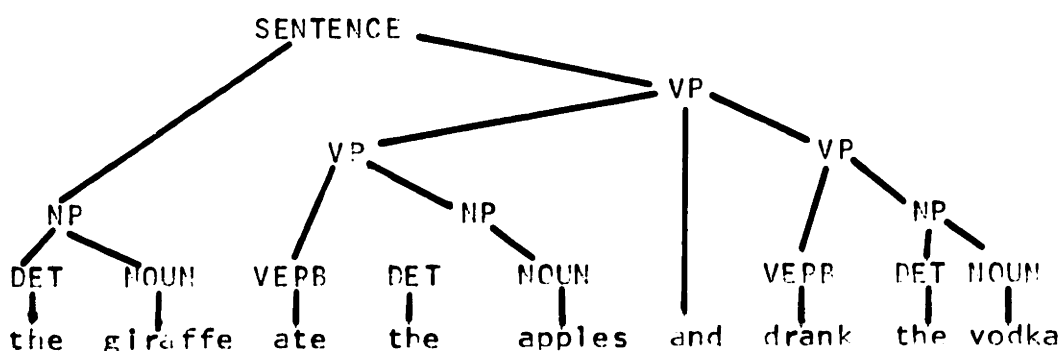
For example, "and" can be defined as a program which is diagrammed:



For example, given the sentence "The giraffe ate the apples and peaches." the program would first encounter "and" after parsing the NOUN apples. It would then try to parse a second NOUN, and would succeed, resulting in the structure:



If we had the sentence, "The giraffe ate the apples and drank the vodka." the parser would first try the same thing. However, "drank" is not a NOUN, so the AND program would fail and the NOUN "apples" would be returned unchanged. This would cause the NP "the apples" to succeed, so the AND program would be called again. It would fail to find a NP beginning with "drank", so the NP "the apples" would be returned, causing the VP to succeed. This time, AND would try to parse a VP and would find "drank the vodka". It would therefore make up a combined VP and cause the entire SENTENCE to be completed with the structure:



The program to actually do this would take only 3 or 4 lines in a PROGRAMMAP grammar. In the actual system, it is more complex as it handles lists (like "A, B, and C") other conjunctions (such as "but") and special constructions (such as "both A and B").

2.4.3 The Dictionary

Since PROGRAMMAP is embedded in LISP, the facilities of LISP for handling atom names are used directly. To define a word, a list of grammatical features is put on its property list under the indicator WORD, and a semantic definition under the indicator SMNTC. Two facilities are included to avoid having to repeat information for different forms of the same word. First, there is an alternate way of defining

words, by using the property indicator WORD1. This indicates that the word given is an inflected form, and its properties are a modified form of the properties of its root. A WORD1 definition has three elements, the root word, the list of features to be added, and the list of features to be removed. For example, we might define the word "go" by: (DEFPROP GO (VERB INTRANSITIVE INFINITIVE) WORD) We could then define "went" as (DEFPROP WENT (GO (PAST)(INFINITIVE)) WORD1) This indicates that the feature INFINITIVE is to be replaced by the feature PAST, but the rest (including the semantic definition) is to remain the same as for "go".

The other facility is an automatic system which checks for simple modifications, such as plurals, "-ing," forms, "-er" and "-est" forms and so forth. If the word as typed in is not defined, the program looks at the way it is spelled, tries to remove its ending (taking into account rules such as changing "running" to "run", but "buzzing" to "buzz"). It then tries to find a definition for the reduced root word, and if it succeeds, it makes the appropriate changes for the ending (such as changing the feature SINGULAR to PLURAL). The program which does this is the one part of the PROGRAMMAR system described here which is specifically built for English. Everything else described is designed

generally for the parsing of any language. In any particular language, this input function would have to be written according to the special rules of morphographic structure. The only requirement for such a program is that its output must be a list, each member of which corresponds to a word in the original sentence, and is in the form described in section 2.4.5. This list is bound to the variable SENT, and is the way in which PROGPARMAP sees its input.

2.4.4 Backup Facilities

As explained in section 2.2.7, there is no automatic backup, but there are a number of special functions which can be used in writing grammars. The simplest, (POPTO X) simply removes nodes from the tree. The argument is a list of features, and the effect is to remove daughters of the currently active node, beginning with the rightmost and working leftward until one is reached with all of those features. (POP X) is the same, except that it also removes the node with the indicated features. If no such node exists, neither function takes any action. (POP) is the same as (POP NIL), and a non-nil value is returned by both functions if any action has been taken.

A very important feature is the CUT variable. One way to do backup is to first try to find the longest possible

constituent at any point, then if for any reason an impasse is reached, to return and try again, limiting the constituent from going as far along in the sentence. For example, in the sentence "Was the typewriter sitting on the cake?", the parser will first find the auxiliary verb "was", then try to parse the subject. It will find the noun group "the typewriter sitting on the cake", which in another context might well be the subject ("the typewriter sitting on the cake is broken."). It then tries to find the verb, and discovers none of the sentence is left. To back up, it must change the subject. A very clever program would look at the structure of the noun group and would realize that the modifying clause "sitting on the cake" must be dropped. A more simple-minded but still effective approach would use the following instructions:

```
(** N PW)
(POP)
((CUT PTW)SUBJECT (EPROP))
```

The first command sets the pointer PTW to the last word in the constituent (in this case, "cake"). The next removes that constituent. The third sets a special pointer, CUT to that location, then sends the program back to the point where it was looking for a subject. It would now try to find a subject again, but would not be allowed to go as far as the word "cake". It might now find "the typerwriter

sitting," an analog to "The man sitting is my uncle." If there were a good semantic program, it would realize that the verb "sit" cannot be used with an inanimate object without a location specified. This would prevent the constituent "the typewriter sitting" from ever being parsed. Even if this does not happen, the program would fail to find a verb when it looked at the remaining sentence, "on the cake." By going through the cutting loop again, it would find the proper subject, "the typewriter," and would continue through the sentence.

Once a CUT point has been set for any active node, no descendant of that node can extend beyond that point until the CUT is moved. Whenever a PROGRAMMAP program is called, the variable END is set to the current CUT point of the node which called it. The CUT point for each constituent is initially set to its END. When the function PARSE is called for a word, it first checks to see if the current CUT has been reached, and if so it fails. The third branch in a three-direction branch statement is taken if the current CUT point has been reached. The CUT pointer is set with the function CUT of one argument, .

2.4.5 Messages

To write good parsing programs, we may at times want to know why a particular PROGRAMMAP program failed, or why a

certain pointer command could not be carried out. In order to facilitate this, two message variables are kept at the top level of the system, MES, and MESP. Messages can be put on MES in two ways, either by using the special failure directions in the branch statements (see section 2.2.5) or by using the functions M and M0, which are exactly like F and F0, except they put the indicated feature onto the message list ME for that unit. When a unit returns either failure or success, MES is bound to the current value of ME, so the calling program can receive an arbitrary list of messages for whatever purpose it may want them. MESP always contains the last failure message received from ** or *.

2.4.6 The Form of the Parsing Tree

Each node is actually a list structure with the following information:

FE	the list of features associated with the node
MB	the place in the sentence where the constituent begins
N	the place immediately after the constituent
H	the subtree below that node (actually a list of its daughters in reverse order, so that H points to the last constituent parsed)
SM	a space reserved for semantic information

These can be used in two ways. If evaluated as variables, they will always return the designated information for the currently active node. C is always a pointer to that node. If used as functions of one argument,

they give the appropriate values for the node pointed to by that argument; so (NB H) gives the location in the sentence of the first word of the last constituent parsed, while (FE(NB H)) would give the feature list of that word.

Each word in the sentence is actually a list structure containing the 4 items:

FE	as above
SMWORD	the semantic definition of the word (see section 2.4.5)
WORD	the word itself (a pointer to an atom)
ROOT	the root of the word (e.g. "run" if the word is "running").

2.4.7 Variables Maintained by the System

There are two types of variables, those bound at the top level, and those which are rebound every time a PROGRAMMAP program is called.

Variables bound at the top level

N	Always points to next word in the sentence to be parsed
SENT	Always points to the entire sentence
PT PTW	Tree and sentence pointers. See Section 2.4.6
MES MESP	List of messages passed up from lower levels. See Section 2.4.9

Special variables bound at each level

C FE NB SM H	See section 2.4.2
NN CUT END	See section 2.4.8. NN always equals (NOT(EQ CUT END))
UNIT	the name of the currently active PROGRAMMAP program
PEST	the list of arguments for the call to PAPSE (These form the initial feature list for the node, but as other features are added, PEST

continues to hold only the original ones.)

T1 T2 T3 Three temporary PROG variables for use by the program in any way needed.

MVB Bound only when a CLAUSE is parsed used as a pointer to the main verb

ME List of messages to be passed up to next level See Section 2.4.9

2.4.8 Pointers

The system always maintains two pointers, PT to a place on the parsing tree, and PTW to a place in the sentence. These are moved by the functions * and ** respectively, as explained in section 2.2.6. The instructions for PT are:

C set PT to the currently active node

H set PT to most recent (rightmost) daughter of C

DL (down-last) move PT to the rightmost daughter of its current value

DLC (down-last completed) like DL, except it only moves to nodes which are not on the push-down list of active nodes.

DF (down-first) like DL, except the leftmost

PV (previous) move PT to its left-adjacent sister

NX (next) move PT to its right-adjacent sister

U (up) move PT to parent node of its current value

N Move PT to next word in sentence to be parsed

The pointer PTW always points to a place in the sentence. It is moved by the function ** which has the same syntax as *, and the commands:

N Set PTW to the next word in the sentence

FW (first-word) set PTW to the first word of the constituent pointed to by PT

LW (last-word) like FW

AW (after-word) like FW, but first word after the constituent

NW (next-word) Set PTW to the next word after its
 current value
PW (previous-word) like NW
SFW (sentence-first-word) set PTW to the first word
 in the sentence
SLW (sentence-last-word) like SFW

Since the pointers are bound at the top level, a program which calls others which move the pointers may want to preserve their location. PTW is a simple variable, and can be saved with a SETQ, but PT operates by keeping track of the way it has been moved, in order to be able to retrace its steps. This is necessary since LISP lists are threaded in only one direction (in this case, from the parent node to its daughters, and from a right sister to its left sister). The return path is bound to the variable PTP, and the command (PTSV X) saves the values of both PT and PTP under the variable X, while (PTPS X) restores both values.

2.4.9 Feature Manipulating

As explained in section 2.2.6, we must be able to attach features to nodes in the tree. The functions F, FQ, and TRNSF are used for putting features onto the current node, while R and RQ remove them. (F A) sets the feature list FE to the union of its current value with the list of features A. (FQ A) adds the single feature A (i.e. it quotes its argument). (TPNSF A B) was explained in Section 2.2.7. R and RQ are inverses of F and FQ. The functions

IS, ISQ, CQ, and NQ are used to examine features. If A points to a node of the tree or word of the sentence, and B points to a feature, (IS A B) returns non-nil if that node has that feature. (ISO A P) is equivalent to (IS A (QUOTE B)), (CQ B) is the same as (ISO C B) (where C always points to the currently active node), and (NQ B) is the same as (ISQ N B) (N always points to the next word in the sentence left to be parsed).

2.5 Comparison with Other Parsers

2.5.1 Older Parsers

When work first began on analyzing natural language with computers, no theories of syntax existed which were explicit enough to be used. The early machine-translator designers were forced to develop their own linguistics as they worked, and they produced rough and ready versions. The parsers were collections of "packaging routines", "inserted structure passes", "labeling subroutines", etc. (see Garvin(16)) which evolved gradually as the grammars were expanded to handle more and more complex sentences. They had the same difficulties as any program designed in this way -- as they became more complex it became harder and harder to understand the interactions within them and to make extensions which were intended to deal with a limited anticipated set of inputs might make it very difficult to extend the system later.

When the machine-translation effort failed, it was clear that it had been premature to try handling all of English without a better background of linguistic theory and an understanding of the mathematical properties of grammars. Computer programs for natural language took two separate paths. The first was to ignore traditional syntax entirely, and to use some sort of more general pattern matching

process to get information out of sentences. Systems such as STUDENT, SIR, ELIZA, and Semantic Memory made no attempt to do complete syntactic analysis of the inputs. They either limited the user to a small set of fixed input forms or limited their understanding to those things they could get while ignoring syntax.

The other approach was to take a simplified subset of English which could be handled by a well-understood form of grammar, such as one of the variations of context-free grammars. There has been much interesting research on the properties of abstract languages and the algorithms needed to parse them. Using this theory, a series of parsing algorithms and representations were developed. For a summary of the computer parsers designed before 1966, see Bobrow (3). A more recent development was Early's context-free parser (13) which operates in a time proportional to the cube of the length of a sentence.

The problem faced by all of these parsers (including the mammoth Harvard Syntactic Analyzer (Kuno and Oettinger (31)) is that such simple models are not adequate for handling the full complexity of natural language. This is discussed theoretically in Chomsky (8) but for our purposes it is more important to note that many aspects which could theoretically be handled would be included only at the

expense of gross inefficiency and unnecessary complexity.

Several people attempted to use Chomsky's transformational grammar as the basis for parsers. (SEE REFERENCES (40) AND (64)) They tried to "unwind" the transformations to reproduce the deep structure of a sentence, which could then be parsed by a context free "base component". It soon became apparent that this was an impossible task. Although transformational grammar is theoretically a "neutral" description of language, it is in fact highly biased toward the process of generating sentences rather than interpreting them. Adapting generation rules to use in interpretation is relatively easy for a context-free grammar, but extremely difficult for transformational grammars. Woods (62) discusses the problems of "combinatorial explosion" inherent in the inverse transformational process. The transformational parsers never went beyond the stage of handling small subsets of English in a very inefficient way.

2.5.2 Augmented Transition Networks

In the past two years, three related parsing systems have been developed to deal with the full complexity of natural language. The first was by Thorne, Bratley, and Dewar (54), and the more recent ones are by Bobrow and Fraser (4) and Woods (62). The three programs operate in

very similar ways, and since woods' is the most advanced and best documented, we will use it for comparison. In his paper Woods compares his system with the other two.

The basic idea of these parsers is the "augmented transition network". The parser is seen as a transition network much like a finite-state recognizer used for regular languages in automata theory.

The first extension is in allowing the networks to make recursive calls to other networks (or to themselves). The condition for following a particular state transition is not limited to examining a single input symbol. The condition on the arc can be something like "NP" where NP is the name of an initial state of another network. This recursively called NP network then examines the input and operates as a recognizer. If it ever reaches an accepting state, it stops, and parsing continues from the end of the NP arc in the original network. These "recursive transition networks" have the power of a context-free grammar, and the correspondence between a network and its equivalent grammar is quite simple and direct.

To parse the full range of natural language, we need a critical addition. Instead of using "recursive transition networks" these parsers use "augmented transition networks", which can "make changes in the contents of a set of

registers associated with the network, and whose transitions can be conditional on the contents of those registers. (Woods (62)). This is done by "adding to each arc of the transition network an arbitrary condition which must be satisfied in order for the arc to be followed, and a set of structure building actions to be executed if the arc is followed."

Augmented transition networks have the power of turing machines (since they have changeable registers and can transfer control depending on the state of those registers). Clearly they can handle any type of grammar which could possibly be parsed by any machine. The advantages lie in the ways in which these augmented networks are close to the actual operations of language, and give us a natural and understandable representation for grammars.

2.5.3 Networks and Programs

How does this type of parser compare with PROGRAMMAP? Is there anything in common between grammars which are networks and grammars which are programs? The reader may have already seen the "joke" in this question. In fact these are just two different ways of talking about doing exactly the same thing!

Let us picture a flowchart for a PROGRAMMAP grammar, in which calls to the function PAPSE are drawn on the arcs

rather than at the nodes. Every arc then is either a request to accept the next word in the input (when the argument of PARSE is a word class), or a recursive call to one of the grammar programs. At each node (i.e. segment of program between conditionals and PARSE calls) we have "a set of arbitrary structure building actions." Our flowchart is just like an augmented transition network.

Now let us picture how Woods' networks are fed to the computer. He uses a notation (see (62) p. 17) which looks very much like a LISP-embedded computer language, such as PPROGRAMMAR or PLANNER. In fact, the networks could be translated almost directly into PLANNER programs (PLANNER rather than LISP or PPROGRAMMAR because of the automatic backup features -- see discussion below).

It is an interesting lesson in computer science to look at Woods' discussion of the advantages of networks, and "translate" them into the advantages of programs. For example, he talks about efficiency of representation. "A major advantage of the transition network model is...the ability to merge the common parts of many context free rules."

Looking at grammars as programs, we can call this "sharing subroutines". He says "The augmented transition network, through its use of flags allows for the merging of

similar parts of the network by recording information in registers and interrogating it...and to merge states whose transitions are similar except for conditions on the contents of the registers." This is the use of subroutines with parameters. In addition, the networks can "capture the regularities of the language...whenever there are two essentially identical parts of the grammar which differ only in that the finite control part of the machine is remembering some piece of information...it is sufficient to explicitly store the distinguishing piece of information in a register and use only a single copy of the subgraph." This is clearly the use of subroutines with an argument!

Similarly we can go through the arguments about efficiency, the ease of mixing semantics with syntax, the ability to include operations which are "natural" to the task of natural language analysis, etc. All of them apply identically whether we are looking at "transition networks" or "programs".

What about "perspicuity"? Woods claims that augmented transition networks retain the perspicuity (ease of reading and understanding by humans) of simpler grammar forms. He says that transformational grammars have the problem that "the effect of a given rule is intimately bound up with its interrelation to other rules...it may require an extremely

complex analysis to determine the effect and purpose." ((62) p.38) This is true, but it would also be true for any grammar complex enough to handle all of natural language. The simple examples of transition networks are indeed easy to read (as are simple examples of most grammars), but in a network for a complete language, the purpose of a given state would be intimately bound up with its interrelation to other states, and the grammar will not be as "perspicuous" as we might hope. This is just as true for programs, but no more so. If we look at the flow chart instead of the listing, programs are equally perspicuous to networks.

If the basic principles are really the same, are there any differences at all between Woods' system and ours? The answer is yes, they differ not in the theoretical power of the parser, but in the types of analysis being carried out.

The most important difference is the theory of grammar being used. All of the network systems are based on transformational grammar. They try to reproduce the "deep structure" of a sentence while doing surface structure recognition. This is done by using special commands to explicitly build and rearrange the deep structures as the parsing goes along. PROGRAMMAR is oriented towards systemic grammar, with its identification of significant features in the constituents being parsed. It therefore emphasizes the

ability to examine the features of constituents anywhere on the parsing tree, and to manipulate the feature descriptions of nodes. In section 2.1 we discussed the advantages of systemic grammar for a language understanding system. Either type of parser could be adapted to any type of grammar, but PROGRAMMAP was specially designed to include "natural" operations for systemic understanding of sentences.

A second difference is in the implementation of special additions to the basic parser. For example in section 2.4.2 we discussed the way in which words like "and" could be defined to act as "demons" which interrupt the parsing at whatever point they are encountered, and start a special program for interpreting conjoined structures. This has many uses, both in the standard parts of the grammar (such as "and") and in handling idioms and unusual structures. If we think in network terms, this is like having a separate arc marked "and" leading from every node in the network. Such a feature could probably be added to the network formulation, but it seems much more natural to think in terms of programs and interrupts.

A third difference is the backup mechanism. The network approach assumes some form of nondeterminism. If there are several arcs leaving a node, there must be some

way to try following all of them. Either we have to carry forward simultaneous interpretations, or keep track of our choices in such a way that the network can automatically revise its choice if the original choice does not lead to an accepting state. This could be done in the program approach by using a language such as PLANNEP with its automatic backup mechanisms. But in section 2.2.7 we discussed the question of whether it is even desirable to do so in handling natural language.

We pointed out the advantage of an intelligent parser which can understand the reasons for its failure at a certain point, and can guide itself accordingly instead of backing up blindly. This is important for efficiency, and Woods is very concerned with ways to modify the networks to avoid unnecessary and wasteful backup by "making the network more deterministic." (see (62) p. 45). It might be interesting to explore a compromise solution in which automatic backup facilities existed, but could be turned on and off. We could do this by giving PPROGRAMAP special commands which would cause it to remember the state of the parsing so that later the grammar could ask to back up to that state and try something else. This is an interesting area for further work on PPROGRAMAR.

It is difficult to compare the performance of different

parsers since there is no standard grammar or set of test sentences. Bobrow and Woods have not published the results of any experiments with a large grammar, but Thorne has published two papers ((54) and (55)) with a number of sample parsings. Our system, with its current grammar of English has successfully parsed all of these examples

3. Inference

3.1 Basic Approach to Meaning

3.1.1 Representing Knowledge

We have described the process of understanding language as a conversion from a string of sounds or letters to an internal representation of "meaning". In order to do this, a language-understanding system must have some formal way to express its knowledge of a subject, and must be able to represent the "meaning" of a sentence in this formalism. The formalism must be structured in such a way that the system can use its knowledge to make deductions, accept new information, answer questions, and interpret commands. Choosing a form for this information is of central importance to both a practical system and a theory of semantics.

First we must decide what kinds of things are to be represented in the formalism. As a beginning, we would like to be able to represent "objects", "properties," and "relations." Later we will have to show how these can be combined to express more complicated knowledge. We will try to find a way to express the meaning of a wide variety of complex sentences.

Using a simple prefix notation, we can represent such facts as "Boise is a city." and "Noah was the father of

Jafeth." as:

(CITY BOISE) (FATHER-OF NOAH JAFETH)

Here, BOISE, NOAH, and JAFETH are specific objects, CITY is a property which objects can have, and FATHER-OF is a relation. It is a practical convenience to list properties and relations first, even though this may not follow the natural English order, so we will do so throughout. Notice that properties are in fact special types of relations which deal with only one object. Properties and relations will be dealt with in identical ways throughout the system. In fact, it is not at all obvious which concepts should be considered properties and which relations. For example, "DeGaulle is old." might be expressed as (OLD DEGAULLE) where OLD is a property of objects or as (AGE DEGAULLE OLD), where AGE is a relation between an object and its age. In the second expression, OLD appears in the position of an object, even though it can hardly be construed as a particular object like BOISE or DEGAULLE. This suggests that we might like to let properties or relations themselves have properties and enter into other relations. This has a deep logical consequence which will be discussed in section 5.1.

In order to avoid confusion, we will need some conventions about notation. Most objects and relationships

do not have simple English names, and those that do often share their names with a range of other meanings. The house on the corner by the market doesn't have a proper name like Jafeth, even though it is just as much a unique object. For the internal use of the system, we will give it a unique name by stringing together a descriptive word and an arbitrary number, then beginning with a colon to remind us it is an object. The house mentioned above might be called :HOUSE374. Properties and relations must also go under an assumed name, since (SHARP X) might mean very different things depending on whether X is a knife or a cheese. We can do the same thing (using a different punctuation mark, #) to represent these two meanings as #SHAPP1 and #SHAPP2. When the meaning intended is clear, we will omit the numbers, but leave the punctuation marks to remind us that it is a property or relation rather than a specific object. Thus, our facts listed above should be written:

(#CITY :BOISE) (#FATHER-OF :NOAH :JAFETH), and either (#OLD :DEGAULLE) or (#AGE :DEGAULLE #OLD).

We are letting properties serve in a dual function -- we can use them to say things about objects (as in "The sky is blue." -- (#BLUE :SKY)) or we can say things about them as if they were objects (as in "Blue is a color." -- (#COLOR #BLUE)). We want to extend this even further, and allow

entire relationships to enter into other relationships. (We distinguish between "relation", the abstract symbol such as #FATHER-OF, and "relationship", a particular instance such as (#FATHER-OF :NOAH :JAFETH)). In accord with our earlier convention about naming things, we can give the relationship a name, so that we can treat it like an object and say (#KNOW :I :REL76) where :REL76 is a name for a particular relationship like (#FATHER-OF :NOAH :JAFETH). We can keep straight which name goes with which relationship by putting the name directly into the relationship. Our example would then become (#FATHER-OF :NOAH :JAFETH :REL76). There is no special reason to put the name last, except that it makes indexing and reading the statements easier. We can tell that :REL76 is the name of this relation, and not a participant since FATHER-OF relates only two objects. Similarly, we knew that it has to be a participant in the relationship (#KNOW :I :REL76) since #KNOW needs two arguments.

We now have a system which can be used to describe more complicated facts. "Harry slept on the porch after he gave Alice the jewels." would become a set of assertions:

```
(#SLEEP :HARRY :PEL1)    (#LOCATION :PEL1 :PORCH)
(#GIVE :HARRY :ALICE :JEWELS :PEL2)    (#AFTER :PEL1 :PEL2)
```

This example points out several facts about the

notation. The number of participants in a relationship depends on the particular relation, and can vary from 0 to any number. We do not need to give every relationship a name -- it is present only if we want to be able to refer to that relationship elsewhere. This will often be done for events, which are a type of relationship with special properties (such as time and place of occurrence).

3.1.2 Philosophical Considerations

Before going on, let us stop and ask what we are doing. In the preceding paragraphs, we have developed a notation for representing certain kinds of meaning. In doing so we have glibly passed over issues which have troubled philosophers and linguists for thousands of years. Countless treatises and debates have tried to analyze just what it means to be an "object" or a "property", and what logical status a symbol such as #BLUE or #CITY should have.

We will not attempt to give a philosophical answer to these questions, but instead take a more pragmatic approach to meaning. Language is a process of communication between people, and is inextricably enmeshed in the knowledge that those people have about the world. That knowledge is not a neat collection of definitions and axioms, complete, concise and consistent. Rather it is a collection of concepts designed to manipulate ideas. It is in fact incomplete,

highly redundant, and often inconsistent. There is no self-contained set of "primitives" from which everything else can be defined. Definitions are circular, with the meaning of each concept depending on the other concepts.

We would like to consider some concepts as "atomic". (i.e. concepts which are considered to have their own meaning rather than being just combinations of other more basic concepts). A property or relation is atomic not because of some special logical status, but because it serves a useful purpose in relation to the other concepts in the speaker's model of the world. For example, the concept #OLD is surely not primitive, since it can be defined in terms of #AGE and number. However, as an atomic property it will often appear in knowledge about people, the way they look, the way they act, etc. Indeed, we could omit it and always express something like "having an age greater than 30", but our model of the world will be simpler and more useful if we have the concept #OLD available instead.

There is no sharp line dividing atomic concepts from non-atomic ones. It would be absurd to have separate atomic concepts for such things as #CITY-OF-POPULATION-23,485 or #PERSON-WEIGHING-BETWEEN-178-AND-181. But it might in fact be useful to distinguish between #BIG-CITY, #TOWN, and #VILLAGE, or between #FAT, and #THIN, since our model may

often use these distinctions.

If our "atomic" concepts are not logically primitive, what kind of status do they have? What is their "meaning"? How are they defined? The answer is again relative to the world-model of the speaker. Facts cannot be classified as "those which define a concept" and "those which describe it." Ask someone to define #PERSON or #JUSTICE, and he will come up with a formula or slogan which is very limited. #JUSTICE is defined in his world-model by a series of examples, experiences, and specific cases. The model is circular, with the meaning of any concept depending on the entire knowledge of the speaker, (not just the kind which would be included in a dictionary). There must be a close similarity between the models held by the speaker and listener, or there could be no communication. If my concept of #DEMOCRACY and yours do not coincide, we may have great difficulty understanding each other's political viewpoints. Fortunately, on simpler things such as #BLUE, #DOG, and #AFTER, there is a pretty good chance that the models will be practically identical. In fact, for simple concepts, we can choose a few primary facts about the concept and use them as a "definition", which corresponds to the traditional dictionary.

Returning to our notation, we see that it is

intentionally general, so that our system can deal with concepts as people do. In English we can treat events and relationships as objects, as in "The war destroyed Johnson's rapport with the people." Within our representation of meaning we can similarly treat an event such as #WAP or a relationship of #PAPPORT in the same way we treat objects. We do not draw a sharp philosophical distinction between "specific objects", "properties", "relationships", "events", etc.

3.1.3 Complex Information

We now have a way to store a data base of assertions about particular objects, properties, and relationships. Next, we want to handle more complex information, such as "All canaries are yellow.", or "A thesis is acceptable if either it is long or it contains a persuasive argument." This could be done using a formal language such as the predicate calculus. Basic logical relations such as implies, or, and, there-exists, etc. are represented symbolically, and information is translated into a "formula". Thus we might have:

```
(FOPALL (X) (IMPLIES(#CANARY X)(#COLOR X #YELLOW)))
```

and

```

(FORALL (X)(IMPLIES
      (AND (#THEESIS X)
            (OR (#LONG X)
                (EXISTS (Y)
                        (AND (#PERSUASIVE Y)
                            (#ARGUMENT Y)
                            (#CONTAINS X Y))))))
      (#ACCEPTABLE X)))

```

Several notational conventions are used. First, we need variables so that we can say things about objects without naming particular ones. This is done with the quantifiers FORALL and EXISTS. Second, we need logical relations like AND, OR, NOT, and IMPLIES. Using this formalism, we can represent a question as a formula to be "proved". To ask "Is Sam's thesis acceptable?" we could give the formula (#ACCEPTABLE :SAM-THEESIS) to a theorem prover to prove by manipulating the formulas and assertions in the data base according to the rules of logic. We would need some additional theorems which would allow the theorem prover to prove that a thesis is long, that an argument is acceptable, etc.

In some theoretical sense, such formulas could express all of our knowledge, but in a practical sense there is something missing. A person would also have knowledge about how to go about doing the deduction. He would know that he should check the length of the thesis first, since he might be able to save himself the bother of reading it, and that

he might even be able to avoid counting the pages if there is a table of contents. In addition to complex information about what must be deduced, he also knows a lot of hints and "heuristics" telling how to do it better for the particular subject being discussed.

Most "theorem-proving" systems do not have any way to include this additional intelligence. Instead, they are limited to a kind of "working in the dark". A uniform proof procedure gropes its way through the collection of theorems and assertions, according to some general procedure which does not depend on the subject matter. It tries to combine any facts which might be relevant, working from the bottom up. In our example given above, we might have a very complex theorem for deciding whether an argument is persuasive. A uniform proof procedure might spend a great deal of time checking the persuasiveness of every argument it knew about, since a clause of the form (PERSUASIVE X) might be relevant to the proof. What we would prefer is a way for a theorem to guide the process of deduction in an intelligent way. Carl Hewitt has worked with this problem and has developed a theorem-proving language called PLANNER. In PLANNER, theorems are in the form of programs, which describe how to go about proving a goal, or how to deduce consequences from an assertion. This is described at length

in section 3.3, and forms the basis for the inference part of our English understander. In PLANNER, our sentence about thesis evaluation could be represented:

```
(DEFINE THEOREM EVALUATE
      ;EVALUATE is the name we are
      ;giving to the theorem

      (THCONSE(X Y)
        ;this indicates the type of
        ;theorem and names its
        ;variables

        (THGOAL(#THEESIS $?X))
          ;show that X is a thesis
          ;the "$?" indicates a variable

        (THOP
          ;THOP is like "or", trying things
          ;in the order given until one works

          (THGOAL(#LONG $?X)(THUSE CONTENTS-CHECK COUNTPAGES))
            ;THUSE says to try the theorem
            ;named CONTENTS-CHECK first,
            ;then if that doesn't work, try
            ;the one named COUNTPAGES

        (THAND
          ;THAND is like "and"

          (THGOAL(#CONTAINS $?X $?Y))
            ;find something Y which is
            ;contained in X

          (THGOAL(#ARGUMENT $?Y))
            ;show that it is an argument

          (THGOAL(#PERSUASIVE $?Y)(THTBF THTRUE))))))
            ;prove that it is persuasive, using
            ;any theorems which are applicable
```

This is similar in structure to the predicate calculus representation given above, but there are important

in section 3.3, and forms the basis for the inference part of our English understander. In PLANNER, our sentence about thesis evaluation could be represented:

```
(DEFINE THEOREM EVALUATE
      ;EVALUATE is the name we are
      ;giving to the theorem

  (THCONSE (X Y)
           ;this indicates the type of
           ;theorem and names its
           ;variables

  (THGOAL (#THEESIS $?X))
           ;show that X is a thesis
           ;the "$?" indicates a variable

  (THOP
           ;THOP is like "or", trying things
           ;in the order given until one works

  (THGOAL (#LONG $?X)(THUSE CONTENTS-CHECK COUNTPAGES))
           ;THUSE says to try the theorem
           ;named CONTENTS-CHECK first,
           ;then if that doesn't work, try
           ;the one named COUNTPAGES

  (THAND
           ;THAND is like "and"

  (THGOAL (#CONTAINS $?X $?Y))
           ;find something Y which is
           ;contained in X

  (THGOAL (#ARGUMENT $?Y))
           ;show that it is an argument

  (THGOAL (#PERSUASIVE $?Y)(THTBF THTRUE))))))
           ;prove that it is persuasive, using
           ;any theorems which are applicable
```

This is similar in structure to the predicate calculus representation given above, but there are important

differences. The theorem is a program, where each logical operator indicates a definite series of steps to be carried out. THGOAL says to try to find an assertion in the data base, or to prove it using other theorems. THUSE gives advice on what other theorems to use, and in what order. THAND and THOR are equivalent to the logical AND and OR except that they give a specific order in which things should be tried. (The "lisping" is to differentiate PLANNEP names from the standard LISP functions AND and OR. This same convention is used for all functions which have LISP analogs.)

The theorem EVALUATE says that if we ever want to prove that a thesis is acceptable, we should first make sure it is a thesis by looking in the data base. Next, we should try to prove that it is long, first by using the theorem CONTENTS-CHECK (which would check the table of contents), and if that fails, by using a theorem named COUNTPAGES (which might in fact call a simple LISP program which thumbs through the paper.) If they both fail, then we look in the data base for something contained in the thesis, check that it is an argument, and then finally try to prove that it is persuasive. Here, we have used (THTBF THTRUE), which is PLANNEP'S way of saying "try anything you know which can help prove it". PLANNEP must then go searching through

all of its theorems on persuasiveness, just as any other theorem prover would. There are two important changes, though. First, we never need to look at persuasiveness at all if we are able to determine that the thesis is long. Second, we only look at the persuasiveness of arguments which we already know are a part of the thesis. We do not get sidetracked into looking at the persuasiveness theorems except for the cases we really want.

PLANNER also does a number of other things, like maintaining a dynamic data base (assertions can be added or removed to reflect the way the world changes in the course of time), allowing us to control how much deduction will be done when new facts are added to the data base, etc. These are all discussed in section 3.3.

3.1.4 Questions, Statements, and Commands

PLANNER is particularly convenient for a language understanding system, since it can express statements, commands, and questions directly. We have already shown how assertions can be stated in simple PLANNER format. Commands and questions are also easily expressed. Since a theorem is written in the form of a procedure, we can let steps of that procedure actually be actions to be taken by a robot. The command "Pick up the block and put it in the box." could be expressed as a PLANNER program:

```
(THAND(THGOAL(#PICKUP :BLOCK23))
      (THGOAL(#PUTIN :BLOCK23 :BOX7)))
```

Remember that the prefix ":" and the number indicate a specific object. The theorems for #PICKUP and #PUTIN would also be programs, describing the sequence of steps to be done.

Earlier we asked about Sam's thesis in English. Now we can ask:

```
(THGOAL (#ACCEPTABLE :SAM-THESIS)(THUSE EVALUATE))
```

Here we have specified that our theorem EVALUATE is to be used. If we evaluated this PLANNER statement, the theorem would be called, and executed just as described on the previous pages. PLANNER would return one of the values "T" or "NIL" depending on whether the statement is true or false.

For a question like "What nations have never fought a war?" PLANNER has the function THEFIND. We would ask:

```
(THEFIND ALL $?X (X Y)
          (THGOAL(#NATION $?X))
          (THINCT
            (THAND(THGOAL(#WAP $?Y))
                  (THGOAL(#PARTICIPATED $?X $?Y))))))
```

and PLANNER would return a list of all such countries.

Using our conventions for giving names to relations and events, we could even ask:

```
(TFIND ALL $?X (X Y Z EVENT)
 (THGOAL(#CHICKEN $?Y))
 (THGOAL(#ROAD $?Z))
 (THGOAL(#CROSS $?Y $?Z $?EVENT))
 (THGOAL(#CAUSE $?X $?EVENT)))
```

There are a number of other functions for answering different types of questions. These will be described in sections 3.3 and 3.4

This brief description has explained the basic concepts underlying the deductive part of our language understanding program. To go with it, we need a complex model of the subject being discussed. This is described in section 3.4. Section 3.3 gives more details about the PLANNER language and its uses.

3.2 Comparison with Previous Programs

In Section 3.1 we discussed ways of representing information and meaning within a language comprehending system. In order to compare our ideas with those in previous systems, we will establish a broad classification of the field. Of course, no set of pigeon-holes can completely characterize the differences between programs, but they can give us some viewpoints from which to analyze different people's work, and can help us see past the superficial differences. We will deal only with the ways that programs represent their information about the subject matter they discuss. Issues such as parsing and semantic analysis techniques are discussed in other sections. We will distinguish four basic types of systems called "special format", "text based", "restricted logic", and "general deductive".

3.2.1 Special Format Systems

Most of the early language understanding programs were of the special format type. Such systems usually use two special formats designed for their particular subject matter -- one for representing the knowledge they keep stored away, and the other for the meaning of the English input. Some examples are: BASEBALL (reference (19)), which stored tables of baseball results and interpreted questions as

"specification lists" requesting data from those tables; SAD SAM (32), which interpreted sentences as simple relationship facts about people, and stored these in a network structure; STUDENT (2), which interpreted sentences as linear equations and could store other linear equations and manipulate them to solve algebra problems; and ELIZA (56), whose internal knowledge is a set of sentence rearrangements and key words, and which sees input as a simple string of words.

These programs all make the assumption that the only relevant information in a sentence is that which fits their particular format. Although they may have very sophisticated mechanisms for using this information (as in CARPS (6), which can solve word problems in calculus), they are each built for a special purpose, and do not handle information with the flexibility which would allow them to be adapted to other uses. Nevertheless, their restricted domain often allows them to use very clever tricks, which achieve impressive results with a minimum of concern for the complexities of language.

3.2.2 Text Based Systems

Some researchers were not satisfied with the limitations inherent in the special-format approach. They wanted systems which were not limited by their construction

to a particular specialized field. Instead they used English text, with all of its generality and diversity, as a basis for storing information. In these "text based" systems, a body of text is stored directly, under some sort of indexing scheme. An English sentence input to the understander is interpreted as a request to retrieve a relevant sentence or group of sentences from the text. Various ingenious methods were used to find possibly relevant sentences and decide which were most likely to satisfy the request.

PROTOSYNTHEX I (48) had an index specifying all the places where each "content word" was found in the text. It tried to find the sentences which had the most words in common with the request (using a special weighting formula), then did some syntactic analysis to see whether the words in common were in the right grammatical relationship to each other. Semantic Memory (41) stored a slightly processed version of English dictionary definitions in which multiple-meaning words were eliminated by having humans indicate the correct interpretation. It then used an associative indexing scheme which enabled the system to follow a chain of index references. An input request was in the form of two words instead of a sentence. The response was the shortest chain which connected them through the associative

index (e.g. if there is a definition containing the words A and B and one containing B and C, a request to relate A and C will return both sentences).

Even with complex indexing schemes, the text based approach has a basic problem. It can only spout back specific sentences which have been stored away, and can not answer any question which demands that something be deduced from more than one piece of information. In addition, its responses often depend on the exact way the text and questions are stated in English, rather than dealing with the underlying meaning.

3.2.3 Limited Logic Systems

The "limited logic" approach attempted to correct these faults of text based systems, and has been used for most of the more recent language understanding programs. First, some sort of more formal notation is substituted for the actual English sentences in the base of stored knowledge. This notation may take many different forms, such as "description lists" (SIP (44)), "kernels" (PROTOSYNTHESIS I(49)) "concept-relation-concept triples" (PROTOSYNTHESIS III (49)), "data nodes" (TLC(42)), "rings" (DEACON(53)), "relational operators" (52), etc. Each of these forms is designed for efficient use in a particular system, but at heart they are all doing the same thing -- providing a

notation for simple assertions of the sort described in section 3.1.1. It is relatively unimportant which special form is chosen. All of the different methods can provide a uniform formalism which frees simple information from being tied down to a specific way of expressing it in English. Once this is done, a system must have a way of translating from the English input sentences into this internal assertion format, and the greatest bulk of the effort in language understanding systems has been this "semantic analysis". We will discuss it at length in chapter 4. For now we are more interested in what can be done with the assertions once they have been put into the desired form.

Some systems (see (42), (52)) remain close to text based systems, only partially breaking down the initial text input. The text is processed by some sort of dependency analysis and left in a network form. Some, such as TLC (42), emphasize the semantic relationships in the network, while others, such as the "Conceptual Parser" (52) remain much closer to the syntactic dependency analysis. What is common to these systems is that they do not attempt to actually answer questions from the stored information. As with text based systems, they try to answer by giving back bits of information directly from the data base. They may have clever ways to decide what parts of the data are

relevant to a request, but they do not try to break the question down and answer it by logical inference. Because of this, they suffer the same deficiencies as text based systems. They have a mass of information stored away, but little way to use it except to print it back out.

Most of the systems which have been developed recently fit more comfortably under the classification "limited logic". In addition to their data base of assertions (whatever they are called), they have some mechanism for accepting more complex information, and using it to deduce the answers to more complex questions. By "complex information" we mean the type of knowledge described in section 3.1.3. This includes knowledge containing logical quantifiers and relationships (such as "Every canary is either yellow or purple," or "If A is a part of B and B is a part of C, then A is a part of C."). By "complex questions", we mean questions which are not answerable by giving out one of the data base assertions, but demand some logical inference to produce an answer.

One of the earliest limited logic programs was SIF (44), which could answer questions using simple logical relations (like the "part" example in the previous paragraph). The complex information was not expressed as data, but was built directly into the SIF operating program.

... the system is designed to answer the question "Is there a bird which is a canary and is a member of the species 'canary'?" The answer is "yes" if the data base contains the assertion "A canary is a bird" and "A canary is a member of the species 'canary'".

Most of the systems which have been developed present the user with a list of the data base assertions. In order to answer a question, the user must select the data base assertions which are relevant to the question. For example, the user may select the assertions "A canary is a bird" and "A canary is a member of the species 'canary'". By "logical information" we mean the type of knowledge described in section 2.1.3. This includes knowledge containing logical relationships or relationships such as "Every canary is a member of the species 'canary'", or "If A is a part of B and B is a part of C, then A is a part of C", or "complex questions", or many questions which are not answerable by giving out one of the data base assertions, but demand some logical inference to produce an answer.

One of the earliest limited logic programs was SIF (41), which could answer questions using simple logical relations (like the "canary" example in the previous paragraph). The complex information was not expressed as data, but was built directly into the SIF operating program.

This meant that the types of complex information it could use were highly limited, and could not be easily changed or expanded. The complex questions it could answer were similar to those in many later limited logic systems, consisting of four basic types. The simplest is a question which translates into a single assertion to be verified or falsified (e.g. "Is John a bagel?") The second is an assertion in which one part is left undetermined (e.g. "Who is a bagel?") and the system responds by "filling in the blank". The third type is an extension of this, which asks for all possible blank-fillers (e.g. "Name all bagels."), and the fourth adds counting to this listing facility to answer count questions (e.g. "How many bagels are there?"). SIR had special logic for answering "how many" questions, using information like "A hand has 5 fingers.", and in a similar way each limited logic system had special built-in mechanisms to answer certain types of questions.

The DEACON (11) system had special "verb tables" to handle time questions, and a bottom-up analysis method which allowed questions to be nested. For example, the question "Who is the commander of the battalion at Fort Fubar?" was handled by first internally answering the question "What battalion is at Fort Fubar?" The answer was then substituted directly into the original question to make it

"Who is the commander of the 69th batallion?", which the system then answered. PROSYNTHEX II (48) had special logic for taking advantage of the transitivity of "is" (e.g. "A boy is a person.", "A person is an animal." therefore "A..."). PROSYNTHEX III (49) and SAFMLAO II (46) bootstrapped their way out of first-order logic by allowing simple assertions about relationships (e.g. "North-of is the converse of South-of."). CONVEPSE (29) converted questions into a "query language" which allowed the form of the question to be more complex but used simple table lookup for finding the answers.

All of the limited logic systems are basically similar, in that complex information is not part of the data, but is built into the system programs. Those systems which could add to their initial data base by accepting English sentences could accept only simple assertions as input. The questions could not involve complex quantified relationships (e.g. "Is there a country which is smaller than every U.S. state?").

3.2.4 General Deductive Systems

These problems of limited logic systems were recognized very early (see Raphael (44) p. 90), and people looked for a more general approach to storing and using complex information. If the knowledge could be expressed in some

standard mathematical notation (such as the predicate calculus), then all of the work logicians have done on theorem proving could be utilized to make an efficient deductive system. By expressing a question as a theorem to be proved (see section 3.1.3), the theorem prover could actually deduce the information needed to answer any question which could be expressed in the formalism. Complex information not easily useable in limited logic systems could be neatly expressed in the predicate calculus, and a body of work already existed on computer theorem proving. This led to the "general deductive" approach to language understanding programs.

The early programs used logical systems less powerful than the full predicate calculus (see (1), (10), and (12)) but the big boost to theorem proving research was the development of the Robinson resolution algorithm (45), a very simple "complete uniform proof procedure" for the first order predicate calculus. This meant that it became easy to write an automatic theorem proving program with two important characteristics. First, the procedure is "uniform" -- we need not (and in fact, cannot) tell it how to go about proving things in a way suited to particular subject matter. It has its own fixed procedure for building proofs, and we can only change the sets of logical

statements (or "axioms") for it to work on. Second, it guarantees that if any proof is possible using the rules of predicate calculus, the procedure will eventually find it (even though it may take a very long time). These are very pretty properties for an abstract deductive system, but the question we must ask is whether their theoretical beauty is worth paying the price of low practicality. We would like to argue that in fact they have led to the worst deficiencies of the theorem-proving question-answerers, and that a very different approach is called for.

The "uniform procedure" approach was adopted by a number of systems (see (17), (18)) as an alternative to the kind of specialized limited logic discussed in the previous section. It was felt that there must be a way to present complex information as data rather than embedding it into the inner workings of the language understanding system. There are many benefits in having a uniform notation for representing problems and knowledge in a way which does not depend on the quirks of the particular program which will interpret them. It enables a user to describe a body of knowledge to the computer in a "neutral" way without knowing the details of the question-answering system, and guarantees that the system will be applicable to any subject, rather than being specialized to handle only one.

Predicate calculus seemed to be a good uniform notation, but in fact it has a serious deficiency. By putting complex information into a "neutral" logical formula, these systems ignored the fact that an important part of a person's knowledge concerns how to go about figuring things out. Our heads don't contain neat sets of logical axioms from which we can deduce everything through a "proof procedure". Instead we have a large set of heuristics and procedures for solving problems at different levels of generality. Of course, there is no reason why a computer should do things the way a person does, but in ignoring this type of knowledge, programs run into tremendous problems of efficiency. As soon as a "uniform procedure" theorem prover gets a large set of axioms (even well below the number needed for really understanding language), it becomes bogged down in searching for a proof, since there is no easy way to guide its search according to the subject matter. In addition, a proof which takes many steps (even if they are in a sequence which can be easily predicted by the nature of the theorem) may take impossibly long since it is very difficult to describe the correct proving procedure to the system.

It is possible to write theorems in a clever way in order to implicitly guide the deduction process, and a

recent paper (Green (17)) describes some of the problems in "techniques for "programming" in first-order logic". First order logic is a declarative rather than imperative language, and to get an imperative effect (i.e. telling it how to go about doing something) takes a good deal of careful thought and clever trickery.

It might be possible to add strategy information to a predicate calculus theorem prover, but with current systems such as QA3, "To change strategies in the current version, the user must know about set-of-support and other program parameters such as level bound and term-depth bound. To radically change the strategy, the user presently has to know the LISP language and must be able to modify certain strategy sections of the program." (Green, p.236(17)). In newer programs such as QA4, there will be a special strategy language to go along with the theorem-proving mechanisms. It will be interesting to see how close these new strategy languages are to PLANNED, and whether there is any advantage to be gained by putting them in a hybrid with a resolution-based system. As to the completeness argument, there are good reasons not to have a complete system -- these are discussed later in this section.

3.2.5 Procedural Deductive Systems

The problem with the limited logic systems wasn't the

fact that they expressed their complex information in the form of programs or procedures. The problem was that these programs were organized in such a way that "...each change in a subprogram may affect more of the other subprograms. The structure grows more awkward and difficult to generalize...Finally the system may become too unwieldy for further experimentation." (Raphael (44) p.91).

Nevertheless, it was necessary to build in more and more of these subprograms in order to accept new subject matter. What was needed was the development of new programming techniques so that systems could retain the capability of using procedural information, but at the same time express this information in a simple and straightforward way which did not depend on the peculiarities and special structure of a particular program or subject of discussion.

A system which partially fits this description is Woods' (63). It uses a quantificational query language for expressing questions, then assumes that there are "semantic primitives" in the form of LISP subroutines which decide such predicates as (CONNECT FLIGHT-23 BOSTON CHICAGO) and which evaluate functions such as "number of stops", "owner", etc. The thing which makes this system different from the limited logic systems is that the entire system was designed without reference to the way the particular "primitive"

functions would operate on the data base. In a way, this is avoiding the issue, since the information which the system was designed to handle (the Official Airline Guide) is particularly amenable to simple table-lookup routines. If we had to handle less structured information of the type usually done with theorem provers, these primitive routines might indeed run into the same problems of interconnectedness described in the quote above, and would become harder and harder to generalize.

PLANNEP was designed by Carl Hewitt as a goal-oriented procedural language to deal with these problems. It has special mechanisms for dealing with assertions in an efficient way, and in addition has the capability to include any complex information which can be expressed in the predicate calculus. More important, the complex information is expressed in the form of procedures, which can include all sorts of knowledge of how to best go about proving things. The language is "goal-oriented", in that we do not have to be concerned about the interaction between the different procedures. If at different places in our knowledge we have theorems which ask whether an object is sturdy (for example in a theorem about support, about building houses, etc.) they are not forced to specify the program which will serve as sturdiness-inspector. Instead

they say something like "Try to find an assertion that X is sturdy, or prove it using anything you can." If we know of special procedures which seem most likely to give a quick answer, we can specify that these should be tried first. But if at some point we add a new sturdiness-tester, we do not need to find out which theorems use it. We need only add it to the data base, and the system will automatically try it (along with any other sturdiness-testers) whenever any theorem gives the go-ahead.

This ability to add new theorems without relating them to other theorems is the advantage of a "uniform" notation. In fact PLANNEP is a uniform notation for expressing procedural knowledge just as predicate calculus is a notation for a more limited range of information. The advantage is that PLANNEP has a hierarchical control structure. In addition to specifying logical relationships, a theorem can take over control of the deduction process. We can have complete control over how the system will operate. In any theorem, we can tell it to try to prove a subgoal using only certain theorems (if we know that the goal is bound to fail unless one of them works), we can tell it to try things in a certain order (and the choice of this order can depend on arbitrarily complex calculations which take place when the subgoal is set up) or

we can even write a "spoiler" theorem, which can tell the system that a goal is certain to fail, and that no other theorems should even be tried.

Notice that this control structure makes it very difficult to characterize the abstract logical properties of PLANNER, such as consistency and completeness. It is worth pointing out here that completeness may in fact be a bad property. It means (we believe, necessarily) that if the theorem-prover is given something to prove which is in fact false, it will exhaust every possible way of trying to prove it before it gives up. By forsaking completeness, we allow ourselves to use good sense in deciding when to give up.

In a truly uniform system, the theorem prover is forced to "rediscover the world" every time it answers a question. Every goal forces it to start from scratch, looking at all of the theorems in the data base (perhaps using some subject-matter-free heuristics to make a rough selection). Because it does not want to be limited to domain-dependent information, it cannot use it at all. PLANNER can operate in this "blindman" mode if we ask it to (and it is less efficient at doing so than a procedure specially invented to operate this way), but it should have to do this only rarely -- when discovering something which was not known or understood when the basic theorems were

written. The rest of the time it can go about proving things which it knows how to do, without a tremendous overhead of having to piece together a proof from scratch each time. As mentioned above, it might be possible to patch "strategy programs" onto theorems in conventional theorem-provers in order to accomplish the same goal. In PLANNER we have the advantage that this can be done naturally using the notation, and the strategy is embedded in the PLANNER theorems, which themselves can be looked at as data. In an advanced system a PLANNER program could be written to learn from experience. Once the "blindman mode" finds a proof, the method it used could be remembered and tried first when a similar goal is generated again. See section 5.1 for more discussion of learning.

To those accustomed to uniform proof procedures, this all sounds like cheating. Is the system really proving anything if you are giving it clues about what to do? Why is it different from a simple set of programmed LISP procedures like those envisioned by Woods? First, the language is designed so that theorems can be written independently of each other, without worrying about when they will be called, or what other theorems and data they will need to prove their subgoals. The language is designed so that if we want, we can write theorems in a form which is

almost identical to the predicate calculus, so we have the benefits of a uniform system. On the other hand, we have the capability to add as much subject-dependent knowledge as we want, telling theorems about other theorems and proof procedures. The system has an automatic goal-tree backup system, so that even when we are specifying a particular order in which to do things, we may not know how the system will go about doing them. It will be able to follow our suggestions and try many different theorems to establish a goal, backing up and trying another automatically if one of them leads to a failure (see section 3.3).

In summary, the main advance in a deductive system using PLANNEP is in allowing ourselves to have a data base of procedures rather than formulas to express complex information. This combines the generality and power of a theorem prover with the ability to accept procedural knowledge and heuristics relevant to the data. It provides a flexible and powerful tool to serve as the basis for a language understanding system. The rest of this chapter describes the PLANNEP language and the way it is used in our system.

3.3 Programming in PLANNER

3.3.1 Basic Operation of PLANNER

The easiest way to understand PLANNER is to watch how it works, so in this section we will present a few simple examples and explain the use of some of its most elementary features.

First we will take the most venerable of traditional deductions:

```
Turing is a human
All humans are fallible
so
Turing is fallible.
```

It is easy enough to see how this could be expressed in the usual logical notation and handled by a uniform proof procedure. Instead, let us express it in one possible way to PLANNER by saying:

```
(THASSERT (HUMAN TURING))
;This asserts that Turing is human.
(DEFPROP THEOREM1
 (THCONSE (X) (FALLIBLE $?X)
 (THGOAL (HUMAN $?X)))
 THEOREM)
;This is one way of saying that all humans
are
;fallible.
```

The proof would be generated by asking PLANNER to evaluate the expression:

```
(THGOAL (FALLIBLE TURING) (THTRF THTRUF))
```

We immediately see several points. First, there are

two different ways of storing information. Simple assertions are stored in a data base of assertions, while more complex sentences containing quantifiers or logical connectives are expressed in the form of theorems.

Second, one of the most important points about PLANNER is that it is an evaluator for statements written in a programming language. It accepts input in the form of expressions written in the PLANNER language, and evaluates them, producing a value and side effects. THASSET is a function which, when evaluated, stores its argument in the data base of assertions or the data base of theorems (which are cross-referenced in various ways to give the system efficient look-up capabilities). A theorem is defined with DEFPROP as are functions in LISP.

In this example we have defined a theorem of the THCONSE type (THCONSE means consequent; we will see other types later). This states that if we ever want to establish a goal of the form (FALLIBLE \$?X), we can do this by accomplishing the goal (HUMAN \$?X), where X is a variable. The strange prefix characters are part of PLANNER's pattern matching capabilities. If we ask PLANNER to prove a goal of the form (A X), there is no obvious way of knowing whether A and X are constants (like TURING and HUMAN in the example) or variables. LISP solves this problem by using the

function QUOTE to indicate constants. In pattern matching this is inconvenient and makes most patterns much bulkier and more difficult to read. Instead, PLANNER uses the opposite convention -- a constant is represented by the atom itself, while a variable must be indicated by adding an appropriate prefix. This prefix differs according to the exact use of the variable in the pattern, but for the time being let us just accept \$? as a prefix indicating a variable. The definition of the theorem indicates that it has one variable, X by the (X) following THCONSE.

The third statement illustrates the function THGOAL, which calls the PLANNER interpreter to try to prove an assertion. This can function in several ways. If we had asked PLANNER to evaluate (THGOAL (HUMAN TUPING)) it would have found the requested assertion immediately in the data base and succeeded (returning as its value some indicator that it had succeeded). However, (FALLIBLE TURING) has not been asserted, so we must resort to theorems to prove it.

Later we will see that a THGOAL statement can give PLANNER various kinds of advice on which theorems are applicable to the goal and should be tried. For the moment, (THTBF THTRUE) is advice that causes the evaluator to try all theorems whose consequent is of a form which matches the goal. (i.e. a theorem with a consequent (\$?Z TUPING) would

be tried, but one of the form (HAPPY \$?Z) or (FALLIBLE \$?Y \$?Z) would not. Assertions can have an arbitrary list structure for their format -- they are not limited to two-member lists or three-member lists as in these examples.) The theorem we have just defined would be found, and in trying it, the match of the consequence to the goal would cause the variable \$?X to be assigned to the constant TUPING. Therefore, the theorem sets up a new goal (HUMAN TUPING) and this succeeds immediately since it is in the data base. In general, the success of a theorem will depend on evaluating a PLANNER program of arbitrary complexity. In this case it contains only a single THGOAL statement, so its success causes the entire theorem to succeed, and the goal (FALLIBLE TUPING) is proved.

Consider the question "Is anything fallible?", or in logic, (EXISTS (Y)(FALLIBLE Y)). This requires a variable and it could be expressed in PLANNER as:

```
(THPROG (Y) (THGOAL (FALLIBLE $?Y)))
```

Notice that THPROG (PLANNER's equivalent of a LISP PROG, complete with GO statements, tags, RETURN, etc.) acts as an existential quantifier. It provides a binding-place for the variable Y, but does not initialize it -- it leaves it in a state particularly marked as unassigned. To answer the question, we ask PLANNER to evaluate the entire THPROG

expression above. To do this it starts by evaluating the THGOAL expression. This searches the data base for an assertion of the form (FALLIBLE \$?Y) and fails. It then looks for a theorem with a consequent of that form, and finds the theorem we defined above. Now when the theorem is called, the variable X in the theorem is identified with the variable Y in the goal, but since Y has no value yet, X does not receive a value. The theorem then sets up the goal (HUMAN \$?X) with X as a variable. The data-base searching mechanism takes this as a command to look for any assertion which matches that pattern (i.e. an instantiation), and finds the assertion (HUMAN TURING). This causes X (and therefore Y) to be assigned to the constant TURING, and the theorem succeeds, completing the proof and returning the value (FALLIBLE TURING).

3.3.2 Backup

There seems to be something missing. So far, the data base has contained only the relevant objects, and therefore PLANNER has found the right assertions immediately. Consider the problem we would get if we added new information by evaluating the statements:

```
(THASSET (HUMAN SOCPATES))
(THASSET (GREEK SOCPATES))
```

Our data base now contains the assertions:

(HUMAN TURING)
 (HUMAN SOCRATES)
 (GREEK SOCRATES)

and the theorem:

(THCONSE (X) (FALLIBLE \$?X)
 (THGOAL (HUMAN \$?X)))

What if we now ask, "Is there a fallible Greek?" In PLANNER we would do this by evaluating the expression:

(THPROG (X) (THGOAL (FALLIBLE \$?X)) (THGOAL (GREEK \$?X)))

THPROG acts like an AND, insisting that all of its terms are satisfied before the THPROG is happy. Notice what might happen. The first THGOAL may be satisfied by the exact same deduction as before, since we have not removed information. If the data-base searcher happens to run into TURING before it finds SOCRATES, the goal (HUMAN \$?X) will succeed, assigning \$?X to TURING. After (FALLIBLE \$?X) succeeds, the THPROG will then establish the new goal (GREEK TURING), which is doomed to fail since it has not been asserted, and there are no applicable theorems. If we think in LISP terms, this is a serious problem, since the evaluation of the first THGOAL has been completed before the second one is called, and the "push-down list" now contains only the THPROG. If we try to go back to the beginning and start over, it will again find TURING and so on, ad infinitum.

One of the most important features of the PLANNEP language is that backup in case of failure is always possible, and moreover this backup can go to the last place where a decision of any sort was made. Here, the decision was to pick a particular assertion from the data base to match a goal. Other decisions might be the choice of a theorem to satisfy a goal, or a decision of other types found in more complex PLANNEP functions such as THOP (the equivalent of LISP OR). PLANNEP keeps enough information to change any decision and send evaluation back down a new path.

In our example the decision was made inside the theorem for FALLIBLE, when the goal (HUMAN \$?X) was matched to the assertion (HUMAN TUPING). PLANNEP will retrace its steps, try to find a different assertion which matches the goal, find (HUMAN SOCRATES), and continue with the proof. The theorem will succeed with the value (FALLIBLE SOCRATES), and the THPROG will proceed to the next expression, (THGOAL (GREEK \$?X)). Since X has been assigned to SOCRATES, this will set up the goal (GREEK SOCRATES) which will succeed immediately by finding the corresponding assertion in the data base. Since there are no more expressions in the THPROG, it will succeed, returning as its value the value of the last expression, (GREEK SOCRATES). The whole course of

the deduction process depends on the failure mechanism for backing up and trying things over (this is actually the process of trying different branches down the subgoal tree.) All of the functions like THCOND, THAND, THOP, etc. are controlled by success vs. failure. Thus it is the PLANNER executive which establishes and manipulates subgoals in looking for a proof.

3.3.3 Differences with Other Theorem-Provers and Languages

Although PLANNER is written as a programming language, it differs in several critical ways from anything which is normally considered a programming language. First, it is goal-directed. Theorems can be thought of as subroutines, but they can be called by specifying the goal which is to be satisfied. This is like having the ability to say "Call a subroutine which will achieve the desired result at this point." Second, the evaluator has the mechanism of success and failure to handle the exploration of the subgoal tree. Other languages, such as LISP, with a basic recursive evaluator have no way to do this. Third, PLANNER contains a bookkeeping system for matching patterns and manipulating a data base, and for handling that data base efficiently.

How is PLANNER different from a theorem prover? What is gained by writing theorems in the form of programs, and giving them power to call other programs which manipulate

data? The key is in the form of the data the theorem-prover can accept. Most systems take declarative information, as in predicate calculus. This is in the form of expressions which represent "facts" about the world. These are manipulated by the theorem-prover according to some fixed uniform process set by the system. PLANNER can make use of imperative information, telling it how to go about proving a subgoal, or to make use of an assertion. This produces what is called hierarchical control structure. That is, any theorem can indicate what the theorem prover is supposed to do as it continues the proof. It has the full power of a general programming language to evaluate functions which can depend on both the data base and the subgoal tree, and to use its results to control the further proof by making assertions, deciding what theorems are to be used, and specifying a sequence of steps to be followed. What does this mean in practical terms? In what way does it make a "better" theorem prover? We will give several examples of areas where the approach is important.

First, consider the basic problem of deciding what subgoals to try in attempting to satisfy a goal. Very often, knowledge of the subject matter will tell us that certain methods are very likely to succeed, others may be useful if certain other conditions are present, while others

may be possibly valuable, but not likely. We would like to have the ability to use heuristic programs to determine these facts and direct the theorem prover accordingly. It should be able to direct the search for goals and solutions in the best way possible, and able to bring as much intelligence as possible to bear on the decision. In PLANNER this is done by adding to our THGOAL statement a recommendation list which can specify that ONLY certain theorems are to be tried, or that certain ones are to be tried FIRST in a specified order. Since theorems are programs, subroutines of any type can be called to help make this decision before establishing a new THGOAL. Each theorem has a name (in our definition at the beginning of Section 3.1.1, the theorem was given the name THEOPEN1), to facilitate referring to it explicitly.

The simplest kind of recommendation is THUSE, which takes a list of theorems (by names) and recommends that they be tried in the order listed. A more general recommendation uses filters which look at the theorem and decide whether it should be tried. The user defines his own filters, accept for the standard filter THTPUE, which accepts any theorem.

The filter command for theorems is THTBF, so a recommendation list of the form:

```
((THUSE TH1 TH2)(THTBF TEST)(THUSE TH-DESPEATION))
```

would mean to first try the theorem named TH1, then TH2, then any theorem which passes the filter named TFST (which the user would define), then if all that fails, use the theorem named TH-DESPEPATION. In our programs, we have made use of only the simple capabilities for choosing theorems -- we do not define filters other than THTRUE. However, there is also a capability for filtering assertions in a similar way, and we do use this, as explained in section 4.3.

3.3.4 Controlling the Data Base

An important problem is that of maintaining a data base with a reasonable amount of material. Consider the first example above. The statement that all humans are fallible, while unambiguous in a declarative sense is actually ambiguous in its imperative sense (i.e. the way it is to be used by the theorem prover). The first way is to simply use it whenever we are faced with the need to prove (FALLIBLE \$?X). Another way might be to watch for a statement of the form (HUMAN \$?X) to be asserted, and to immediately assert (FALLIBLE \$?X) as well. There is no abstract logical difference, but the impact on the data base is tremendous. The more conclusions we draw when information is asserted, the easier proofs will be, since they will not have to make the additional steps to deduce these consequences over and over again. However since we don't have infinite speed and

size, it is clearly folly to think of deducing and asserting everything possible (or even everything interesting) about the data when it is entered. If we were working with totally abstract meaningless theorems and axioms (an assumption which would not be incompatible with many theorem-proving schemes), this would be an insoluble dilemma. But PLANNER is designed to work in the real world, where our knowledge is much more structured than a set of axioms and rules of inference. We may very well, when we assert (LIKES \$?X POETRY) want to deduce and assert (HUMAN \$?X), since in deducing things about an object, it will very often be relevant whether that object is human, and we shouldn't need to deduce it each time. On the other hand, it would be silly to assert (HAS-AS-PART \$?X SPLEEN), since there is a horde of facts equally important and equally limited in use. Part of the knowledge which PLANNER should have of a subject, then, is what facts are important, and when to draw consequences of an assertion. This is done by having theorems of an antecedent type:

```
(DEFPROP THEOREM2
  (THANTE (X Y) (LIKES $?X $?Y)
           (THASSEPT (HUMAN $?X)))
  THEOREM)
```

This says that when we assert that X likes something, we should also assert (HUMAN \$?X). Of course, such theorems

do not have to be so simple. A fully general PLANNEP program can be activated by an THANTE theorem, doing an arbitrary (that is, the programmer has free choice) amount of deduction, assertion, etc. Knowledge of what we are doing in a particular problem may indicate that it is sometimes a good idea to do this kind of deduction, and other times not. As with the CONSEQUENT theorems, PLANNEP has the full capacity when something is asserted, to evaluate the current state of the data and proof, and specifically decide which ANTFCEDENT theorems should be called.

PLANNEP therefore allows deductions to use all sorts of knowledge about the subject matter which go far beyond the set of axioms and basic deductive rules. PLANNEP itself is subject-independent, but its power is that the deduction process never needs to operate on such a level of ignorance. The programmer can put in as much heuristic knowledge as he wants to about the subject, just as a good teacher would help a class to understand a mathematical theory, rather than just telling them the axioms and then giving theorems to prove.

3.3.5 Events and States

Another advantage in representing knowledge in an imperative form is the use of a theorem prover in dealing

with processes involving a sequence of events. Consider the case of a robot manipulating blocks on a table. It might have data of the form, "block1 is on block2," "block2 is behind block3", and "if x is on y and you put it on z, then x is on z, and is no longer on y unless y is the same as z". Many examples in papers on theorem provers are of this form (for example the classic "monkey and bananas" problem). The problem is that a declarative theorem prover cannot accept a statement like (ON B1 B2) at face value. It clearly is not an axiom of the system, since its validity will change as the process goes on. It must be put in a form (ON B1 P2 S0) where S0 is a symbol for an initial state of the world. See (Green (17)) for a discussion of such "state" problems.

The third statement might be expressed as:

```
(FORALL (X Y Z S)(AND (ON X Y (PUT X Y S))
                      (OR(EQUAL Y Z)
                         (NOT(ON X Z (PUT X Y S))))))
```

In this representation, PUT is a function whose value is the state which results from putting X on Y when the previous state was S. We run into a problem when we try to ask (ON Z W (PUT X Y S)) i.e. is block Z on block W after we put X on Y? A human knows that if we haven't touched Z or W we could just ask (ON Z W S) but in general it may take a complex deduction to decide whether we have actually moved them, and even if we haven't, it will take a whole chain of

deductions (tracing back through the time sequence) to prove they haven't been moved. In PLANNER, where we specify a process directly, this whole type of problem can be handled in an intuitively more satisfactory way by using the primitive function THEPASE.

Evaluating (THEPASE (ON \$?X \$?Y)) removes the assertion (ON \$?X \$?Y) from the data base. If we think of theorem provers as working with a set of axioms, it seems strange to have function whose purpose is to erase axioms. If instead we think of the data base as the "state of the world" and the operation of the prover as manipulating that state, it allows us to make great simplifications. Now we can simply assert (ON B1 B2) without any explicit mention of states. We can express the necessary theorem as:

```
(DEFPROP THEOREM'S
  (THCONSE (X Y Z) (PUT $?X $?Y)
    (THGOAL (ON $?X $?Z))
    (THEPASE (ON $?X $?Z))
    (THASSERT (ON $?X $?Y)))
  THEOREM'S)
```

This says that whenever we want to satisfy a goal of the form (PUT \$?X \$?Y), we should first find out what thing Z the thing X is sitting on, erase the fact that it is sitting on Z, and assert that it is sitting on Y. We could also do a number of other things, such as proving that it is indeed possible to put X on Y, or adding a list of specific

instructions to a movement plan for an arm to actually execute the goal. In a more complex case, other interactions might be involved. For example, if we are keeping assertions of the form (ABOVE $?X$ $?Y$) we would need to delete those assertions which became false when we erased (ON $?X$ $?Z$) and add those which became true when we added (ON $?X$ $?Y$). ANTECEDENT theorems would be called by the assertion (ON $?X$ $?Y$) to take care of that part, and a similar group called EPASING theorems can be called in an exactly analogous way when an assertion is erased, to derive consequences of the erasure. Again we emphasize that which of such theorems would be called is dependent on the way the data base is structured, and is determined by knowledge of the subject matter. In this example, we would have to decide whether it was worth adding all of the ABOVE relations to the data base, with the resultant need to check them whenever something is moved, or instead to omit them and take time to deduce them from the ON relation each time they are needed.

Thus in PLANNER, the changing state of the world can be mirrored in the changing state of the data base, avoiding any need to make explicit mention of states, with the requisite overhead of deductions. This is possible since the information is given in an imperative form, specifying

BE

WIT

theorems as a series of specific steps to be

If we look back to the distinction between theorems and theorems made on the first page, it would have established that the base of assertions is "state of the world", while the base of theorems is "permanent knowledge of how to deduce things". This is not exactly true, and one of the most interesting possibilities in PLANNER is the capability for itself to create and modify the PLANNER function up the theorem base. Rather than simply making a particular PLANNER function might be written together a new theorem or make changes to an existing theorem, in a way dependent on the data and knowledge. It seems likely that meaningful behavior involves this type of behavior rather than simply adding parameters or adding more individual facts (a declarative data base).

3.3.6 PLANNER Functions

There are a number of other PLANNER components to suit a range of problem-solving needs. They are described in detail in AI Memos ??? and ??? and (???). We will describe only those which are used in our question answering program and which we will refer to later.

We have already mentioned the basic functions and described how they operate. THGOAL looks for assertions in the data base, and calls theorems to achieve goals. THAND takes a list of PLANNER expressions and succeeds only if they all succeed in the order they are listed. THOP takes a similar list and tries the expressions in order, but succeeds as soon as one of them does. Remember that in case of a failure farther along in the deduction, THOP can take back its decision and continue on down the list. The other simple LISP functions PPOG, COND, and NOT have their PLANNER analogs, THPPOG, THCOND, and THNOT, which operate just as their LISP counterparts, except that they are controlled by the distinction between "failure" and "success" instead of the distinction between NIL and non-NIL.

One of the most useful functions is THEIND, which is used to find all of the objects or assertions satisfying a PLANNER condition. THEIND keeps trying different assignments of variables, and each time it succeeds, it adds the result to a list. When it is done, it returns this list. For example, if we want to find all of the red blocks, we can evaluate

```
(THEIND ALL $?X (X)
      (THGOAL(BLOCK $?X))
      (THGOAL(COLOR $?X RED)))
```

The function THFIND takes four kinds of information. First, there is a parameter, telling it how many objects to look for. When we use ALL, it looks for as many as it can find, and succeeds if it finds any. If we use an integer, it succeeds as soon as it finds that many, without looking for more. If we want to be more complex, we can tell it three things: a. how many it needs to succeed; b. how many it needs to quit looking, and c. whether to succeed or fail if it reaches the upper limit set in b.

Thus if we want to find exactly 3 objects, we can use a parameter of (3 4 NIL), which means "Don't succeed unless there are three, look for a fourth, but if you find it, fail".

The second bit of information tells it what we want in the list it returns. For our purposes, this will always be the variable name of the object we are interested in. The third item is a list of variables to be used, and the fourth is the body of the THFIND statement. It is this body that must be satisfied for each object found.

3.4 The BLOCKS World

We need a subject to discuss with our language-understanding program which gives us many different types of things to say and in which we can carry on a discourse, containing statements, questions, and commands. We have chosen to pretend we are talking to a very simple type of robot (like the ones being developed in AI projects at Stanford and MIT) with only one arm and an eye. It can look at a scene containing toy objects like blocks and balls, and can manipulate them with its hand.

We have not tried to use an actual robot or to simulate it in physical detail. Since we are interested primarily in complex language activity, we have adopted a very simplified model of the world, and the "robot" exists only as a display on the CRT scope attached to the computer.

3.4.1 Objects

First we must decide what objects we will have in the world. In 3.1, we adopted some conventions for notation in representing objects and assertions. Any symbol which begins with ":" represents a specific object, while anything beginning with "#" is the name of a property or relation.

The model begins with the two participants in the discussion, the robot (named :SHPDLU), and the person (called :FRIEND). The robot has a hand (:HAND), and

manipulates objects on a table (:TABLE), which has on it a box (:BOX). The rest of the physical objects are toys -- blocks, pyramids, and balls. We give them the names :B1, :B2, :B3,...

Next we must decide on the set of concepts we will use to describe these objects. We can represent these in the form of a tree:

```

          !#TABLE
          !
          !#BOX      !#BLOCK
          !          !
!#PHYSOB--!#MANIP---!#BALL
!          !          !
!#ROBOT   !#HAND    !#PYRAMID
!          !          !
!#PERSON  !#STACK

```

The symbol #PHYSOB stands for "physical object", and #MANIP for "manipulable object" (i.e. something the robot can pick up).

We could use these as simple predicates, and have assertions like (#ROBOT :SHRDLU), (#HAND :HAND), and (#PYRAMID :B5) to say that Shrdlu is a robot, the hand is a hand, and :B5 is a pyramid. In section 4.4.3, we describe the way the language programs choose an English phrase to describe an object. In order to do so, they need a basic noun -- the one we would use to say "this is a ...". If we represented the concepts in the above tree using simple predicates, and then used the same form for other

predicates, such as colors (for example, (#BLUE :P5)), the language generating routines would have no easy way to know which was the "basic" property. It would be necessary to keep lists and continually check. Instead, we adopt a different way of writing these concepts. We use the concept #IS to mean "has as its basic description", and write (#IS :SHPDLU #ROBOT), (#IS :HAND #HAND), and (#IS :B5 #PYRAMID).

Looking at the tree, we see that the properties #PHYSOB and #MANIP cannot be represented in this fashion, since any object having them also has a basic description. We therefore write (#MANIP :B5) and (#PHYSOB :TABLE).

Next, we would like to assign physical properties to these objects, such as size, shape, color, and location. Shape and color are handled with simple assertions like (#COLOR :BOX #WHITE) and (#SHAPE :B5 #POINTED). The possible shapes are #ROUND, #POINTED, AND #RECTANGULAR, and the colors are #BLACK, #RED, #WHITE, #GREEN, and #BLUE. Of course it would involve no programming to introduce other shape or color names -- all that we would do is use them in an assertion, like (#COLOR :B11 #MAUVE), and add an assertion telling what type of thing they are. The property names themselves can be seen as objects, and we have the concepts #COLOR and #SHAPE, to make assertions like (#IS #BLUE #COLOR), and (#IS #RECTANGULAR #SHAPE).

Size and location are more complex, as they depend on the way we choose to represent physical space. We have adopted a standard three-dimensional coordinate system, with coordinates ranging from 0 to 1200 in all three directions. (The number 1200 was chosen for convenience in programming the display). The coordinate point (0 0 0) is in the front lower left-hand corner of the scene.

We have made the simplifying assumption that objects are not allowed to rotate, and therefore always keep their orientation aligned with the coordinate axes. We can represent the position of an object by giving the coordinates of its front lower left-hand corner, and can specify its size by giving the three dimensions. We use the symbols #SIZE and #AT, and put the coordinate triples as a single element in the assertions. For example, we might have (#AT :B5 (400 600 200)), and (#SIZE :B5 (100 100 300)).

Since we assume that the robot has an eye, the system begins the dialog with complete information about the objects in the scene, their shapes, sizes, colors, and locations. In addition to the PLANNER assertions, the system keeps a table of sizes and locations for more efficient calculation when looking for empty spaces.

3.4.2 Relations

The basic relations we will need for this model are the

spatial relations between objects. Since we are interested in moving objects around in the scene, one of the most important relations is #SUPPORT. The initial data base contains all of the applicable support relations for the initial scene, and every time an object is moved, an antecedent theorem removes the old assertion about what was supporting it, and puts in the correct new one. We have adopted a very simplified notion of support, in which an object is supported by whatever is directly below its center of gravity, at the level of its bottom face. Therefore, an object can support several others, but there is only one thing supporting it. Of course this is an extreme simplification since it does not even recognize that a simple bridge is supported. If this program were to be adapted to use with an actual robot, a much more general idea of support would be necessary. Along with the #SUPPORT relations, we keep track of the property #CLEARTOP. The assertion (#CLEARTOP X) will be in the data base if and only if there is no assertion (#SUPPORT X Y) for any object Y. It is also kept current by antecedent theorems.

A second relation which is kept in the data base is #CONTAIN. The first participant must be the box, since this is the only container in the scene. The information about what is contained in the box is also kept current by

an antecedent theorem. The relation #GRASPING is used to indicate what object (if any) the robot's hand is grasping. It is theoretically a two-place predicate, relating a grasper and a graspee, as in (#GRASPING :SHPDLU :P2). Since there is only one hand in our scene, it is clear who must be doing the grasping, so the assertion is reduced to (#GRASPING :B2).

The other relation which is stored in the data base is the #PAPT relation between an object and a stack. We can give a name to a stack, such as :S1, and assert (#PAPT :B2 :S1). As objects are moved, the changes to the data base are again made automatically by antecedent theorems which notice changes of location.

As we explained in section 3.3.3, we must decide what relations are useful enough to occupy space in our data base, and which should be recomputed from simpler information each time we need them. We have included relations like #SUPPORT and #CONTAIN because they are often referenced in deciding how to move objects. We can think of other relations, such as the relative position of two objects, which can be computed from their locations, and are not used often enough to be worth keeping in the data base and partially recomputing every time something is moved. We represent these relations using the symbols #RIGHT, #BEHIND,

and #ABOVE. (These represent the direction of the positive coordinate axis for X, Y, and Z respectively). We do not need the converse relations, since we can represent a fact like ":B1 is below :B2" by (#ABOVE :B2 :B1), and our semantic system can convert what is said to this standard format. The symbol #ON is used to represent the transitive closure of #SUPPORT. That is, Z is #ON A if: A supports B, B supports C, ... supports Z.

The three spatial relations use a common consequent theorem called TC-LOC which decides if they are true by looking at the coordinates and sizes of the objects. The #ON relation has a consequent theorem TC-ON which looks for chains of support. (Notice that the prefix TC- stands for Theorem Consequent, and is attached to all of our consequent theorems. Similarly, TA- and TE- are used for antecedent and erasing theorems.)

The measurements of #HEIGHT, #WIDTH, and #LENGTH are represented as a simple assertion, like (#HEIGHT :B3 100), but they are not stored in the data base. They are computed when needed from the #SIZE assertion, and can be accessed by using the theorem TC-MEASURE, or by using a functional notation. The expression (#HEIGHT X) evaluates to the height of whatever object the variable X is bound to. If #SIZE is used in this way, it returns a measure of "overall

size" to be used for comparisons like "bigger". Currently it returns the sum of the X, Y, and Z coordinates, but it could be easily changed to be more in accord with human psychology.

In order to compare measurements, we have the relation #MOPE. The sentence ":B1 is shorter than :B2" is equivalent to the assertion (#MOPE #HEIGHT :B2 :B1). Again, we do not need the relation "less" since we can simply reverse the order of the objects. The relation #ASMUCH is used in the same way, to express "greater than or equal", instead of "strictly greater than". None of these assertions are stored (if we have ten objects, there will be almost 400 relationships), but are computed from more basic information as they are needed.

One final relationship is #LIKE, which relates a person or robot to any object. There is a theorem which shows that the robot likes everything, but knowledge about what the human user likes is gathered from his statements. The semantic programs can use statements about liking to generate further PLANNEP theorems which are used to answer questions about what :FRIEND likes.

3.4.3 Actions

The only events that can take place in our world are actions taken by the robot in moving its hand and

manipulating objects. At the most basic level, there are only three actions which can occur -- MOVETO, GPASP, and UNGRASP. These are the actual commands sent to the display routines, and could theoretically be sent directly to a physical robot system.

The result of calling a consequent theorem to achieve a goal requiring motion, like (#PUTON :B3 :B4), is a plan -- a list of instructions using the three elementary functions. MOVETO moves the hand and whatever it is currently grasping to a set of specified coordinates. GPASP sets an indicator that the grasped object is to be moved along with the hand, and UNGRASP unsets it. The robot grasps by moving its hand directly over the center of the object on its top surface, and turning on a "magnet". It can do this to any manipulable object, but can only grasp one thing at a time. Using these elementary actions, we can build a hierarchy of actions, including goals which may involve a whole sequence of deductions and actions, like #STACKUP.

The semantic programs never need to worry about things like physical coordinates or specific motion instructions, but can produce input for higher-level theorems which do the detailed work.

At a slightly higher level, we have the PLANEP concepts #MOVEHAND, #GRASP and #UNGPASP, and corresponding

consequent theorems to achieve them. There is a significant difference between these and the functions listed above. Calling the function MOVE TO actually causes the hand to move. On the other hand, when PLANNER evaluates a statement like:

```
(THGOAL(#MOVEHAND (600 200 300))(THUSF TC-MOVEHAND))
```

nothing is actually moved. The theorem TC-MOVEHAND is called, and it creates a plan to do the motion, but if this move causes us to be unable to achieve a goal at some later point, the PLANNER backup mechanism will automatically erase it from the plan. The robot plans the entire action before actually moving anything, trying all of the means it has to achieve its goal.

The theorems also do some checking to see if we are trying to do something impossible. For example, TC-MOVEHAND makes sure the action would not involve placing a block where there is already an object, and TC-UNGPASP fails unless there is something supporting the object it wants to let go of.

TC-GRASP is more complex, as it can cause a series of actions. If PLANNER tries the goal:

```
(THGOAL (#GPASP :B1)(THUSE TC-GPASP))
```

the theorem TC-GPASP can do a number of things. It checks to make sure :B1 is a graspable object by looking in the

data base for (#MANIP :B1). If the hand is already in the right position to grasp the object, TC-GRASP can then add the command (GRASP (QUOTE :B1)) to the plan. If not, it must first get the hand there. This may involve complications -- the hand may already be holding something, or there may be objects sitting on top of the one it wants to grasp. In the first case, it must get rid of whatever is in the hand, using the the command #GET-RID-OF. The easiest way to get rid of something is to set it on the table, so TC-GET-RID-OF creates the goal (#PUTON \$?X :TABLE), where the variable \$?X is bound to the object the hand is holding. TC-PUTON must in turn find a big enough empty place to set down its burden, using the command #FINDSPACE, which performs the necessary calculations, using information about the sizes and locations of all the objects. TC-PUTON then creates a goal using #PUT, which calculates where the hand must be moved to get the object into the desired place, then calls #MOVEHAND to actually plan the move. If we look at the logical structure of our active goals at this point, assuming that we want to grasp :P1, but were already grasping :B2, we see:

```
(#GRASP :B1)
  (#GET-RID-OF :B2)
    (#PUTON :B2 :TABLE)
      (#PUT :B2 (453 201 0))
        (#MOVEHAND (553 301 100))
```

After moving, TC-PUTON calls #UNGPASP, and we have achieved the first part of our original goal -- emptying the hand. Now we must clear off the block we want to grasp. TC-GRASP sets up the goal:

```
(THGOAL(#CLEARTOP :B2)(THUSE TC-CLEAPOP))
```

This is a good example of the double use of PLANNER goals to both search the data base and carry out actions. If the assertion (#CLEARTOP :B1) is present, it satisfies this goal immediately without calling the theorem. However if :B1 is not already clear, this THGOAL statement calls TC-CLEAPOP which takes the necessary actions.

TC-CLEAPOP will try to #GET-PID-OF the objects on top of :B1. This will in turn use #PUTON, which uses #PUT. But TC-PUT may have more to do this time, since the hand is not already grasping the object it has to move. It therefore sets up a goal to #GPASP the object, calling TC-GPASP. We have gone full circle, and are back in the theorem we started in. Of course this is no problem in a recursive language, and the process continues as it should until all of the goals have been achieved, or all of the methods included in the theorems have failed.

We have gone through this example in some detail to give a feeling for the goal-oriented programs used by

PLANNER. The programs are highly recursive, with a clear subgoal structure, and with theorems often calling themselves to achieve subgoals. During all of this, PLANNER is keeping track of what is being done in such a way that it can back up and try something different if necessary. For example, if TC-GET-RID-OF puts an object on the table, and it later is in the way of something which must be done, a failure will propagate back to TC-GET-RID-OF, and it will try putting it somewhere else.

We can list the different action concepts more systematically, showing the form of the goal statements, and the actions taken by the theorems corresponding to them. All of these goals can be used internally within the BLOCKS system, and most of them can be called directly by the semantic programs, as direct translations of English commands. Some, like #MOVEHAND cannot be called by the linguistic programs, since the semantic routines do not include a way to specify exact coordinates in English.

BLOCKS commands

(#MOVEHAND (X Y Z))	Move the center of the hand to location (X Y Z). Anything being grasped goes along automatically
(#UNGPASP)	Let go of whatever the hand is holding. Fails if the object is not supported
(#GPASP X)	Grasp object X, doing any manipulations necessary to get to it
(#PUT W (X Y Z))	Put W at location (X Y Z). Fails unless the space is empty

(#RAISEHAND) Raise the hand and whatever it holds as high as it will go

(#PICKUP X) Grasp X and raise the hand

(#PUTON X Y) Put object X on object Y. If there is not enough free space on Y, move objects to make it

(#PUTIN X Y) The same as #PUTON, except that Y must be a box, and different methods are used to find space

(#GET-RID-OF X) Try to put X on the table, and if that fails put it on anything else

(#CLEARTOP X) Get rid of everything which is on X

(#STACKUP (X Y...)) Stack X, Y, ... on top of each other, in order of size

(#FINDSPACE A (X Y Z) B $_C$) This goal can be achieved by two different theorems. TC-FINDSPACE tries to find a space of size (X Y Z) on top of object A, counting any space occupied by B as empty. $_C$ is a variable binding used to return the answer. If this fails, TC-MAKESPACE can create the space by moving objects

3.4.4 Memory

In order to answer questions about past events, the BLOCKS programs remember selected parts of their subgoal tree. They do this by creating objects called events, and putting them on an EVENTLIST. The system does not remember the detailed series of specific steps like #MOVEHAND, but keeps track of the larger goals like #PUTON and #STACKUP. The time of events is measured by a clock which starts at 0 and is incremented by 1 every time any motion occurs. The theorems which want to be remembered use the functions MEMORY and MEMOREND, calling MEMOPY when the theorem is entered and MEMOPEND when it exits. MEMOPEND causes an

event to be created, combining the original goal statement with an arbitrary name (chosen from E1, E2,...). If we call TC-PUTON with the goal (#PUTON \$?X \$?Y), with the variables X and Y bound to :B1 and :B2 respectively, the resulting event which is put into the data base is (#PUTON E1 :B1 :B2). The event name is second for unimportant technical reasons.

In addition to putting this assertion in the data base, MEMOEND puts information on the property list of the event name -- the starting time, ending time, and reason for each event. The reason is the name of the event nearest up in the subgoal tree which is being remembered. The reason for goals called by the linguistic part of the system is a special symbol meaning "because you asked me to". MEMORY is called at the beginning of a theorem to establish the start time and declare that theorem as the "reason" for the subgoals it calls.

A second kind of memory keeps track of the actual physical motions of objects, noting each time one is moved, and recording its name and the location it went to. This list can be used to establish where any object was at any past time.

When we want to pick up block :B1, we can say: (THGOAL(#PICKUP :B1)), and it is interpreted as a command.

How can we ask "Did you pick up :B1?"? When the robot picked it up, an assertion like (#PICKUP E2 :B1) was stored in the data base. Therefore if we ask PLANNEP

```
(THIPPOG(X)
  (THGOAL (#PICKUP $?X :B1)))
```

it will find the assertion, binding the variable X to the event name E2. Since the property list of F2 gives its starting and ending times, and its reason, this is sufficient information to answer most questions. If we want to ask something like "Did you pick up :B1 before you built the stack?" we need some way to look for particular time intervals. This is done by using a modified version of the event description, including a time indicator. The exact form of the time indicator is described in the section on semantics, but the way it is used to establish a goal is:

```
(THGOAL(#PICKUP $?X :B1 $?TIME)(THUSE TCTE-PICKUP))
```

The prefix TCTE- on the name of a theorem means that it includes a time and an event name. Ordinarily when such a theorem is entered, the variable TIME would have a value, while the variable X would not. The theorem looks through the data base for stored events of the form (#PICKUP \$?X :B1) and checks them to see if they agree with the time TIME.

For some events, like #PUTON, this is sufficient since

the system remembers every #PUTON it does. For others, like #PICKUP less information is kept. When #PICKUP is called as a goal at the top level, it is remembered. But the system does not remember each time something was picked up in the course of moving the toys around. The fact that a block was picked up can be deduced from the fact that it was put somewhere, and the theorem TCTE-PICKUP actually looks at a number of different events to find all the occasions on which an object was really picked up.

For relations, we also need to be able to include time, for example, "Was the block behind the pyramid before...?" In this case, no assertions are stored, since the memory of motion events is sufficient to reconstruct the scene. There are special theorems with the prefix TCT- which try to verify a relation with a time condition. For example, we can ask "Is :B1 on :B2?" with the goal

```
(THGOAL(#ON :P1 :B2)(THUSE TC-ON))
```

To ask "Was :B1 on :B2 before...?" we bind the variable TIME to the representation of the time we are interested in, and ask

```
(THGOAL(#ON :B1 :B2 $?TIME)(THUSE TCT-ON))
```

The theorem TCT-ON is the same as TC-ON except that it deals with the specified time instead of the present. Similar TCT- theorems exist for all of the spatial relations, and

for properties which change in time, such as #CLEAPTCP and #AT.

4. Semantics

4.1 What is Semantics?

4.1.1 the Province of Semantics

The field of semantics has always been a hazy swampland. There is little agreement among "semanticists" where its borders lie or what the terrain looks like. Logicians, philosophers, and linguists all approach it with the tools of their own trade, and the problem of just defining "semantics" and "meaning" have occupied volumes of debate.

In trying to program computers to understand natural language, it has been necessary to have a much more explicit and complete notion of semantics. The attempts at writing language understanding programs have made it much more clear just what a semantic theory has to do, and how it must connect with the syntactic and logical aspects of language. In practical terms, we need a transducer which can look at the results of syntactic analysis, and from this produce data which is acceptable to the logical deductive system.

In the preceding chapters we have described the two ends of a language system -- a syntactic parser with a grammar of English, and a deductive system with a base of knowledge about a particular subject. What does our semantic theory have to do to fill the gap? We can see that

it needs to operate at three levels.

First, there must be a way to define the meanings of words. We pointed out in the section on "meaning" (section 3.1) that the real "meaning" of a word or concept cannot be defined in simple dictionary terms, but involves its relationship to our entire vocabulary and structure of concepts. However, we can talk about the formal description we need to attach to a word which allows it to be integrated into the system. In the rest of this chapter, we will use the word "meaning" in this more limited sense, describing those formal aspects of the meaning of a word (or syntactic construction) which are attached to it as its dictionary definition.

We should have a formalism which does not depend on the details of the semantic programs, but which allows users to add to the vocabulary in a simple and natural way. It should also be possible to handle all of the quirks and idiosyncracies of meaning which words can have, instead of limiting ourselves to "well-behaved" standard words.

At the next level we must relate the meanings of the words in a sentence to each other and to the meaning of the syntactic structures. We need an analysis of the ways in which English structures are designed to convey meaning, and what role the different words play in this meaning.

Finally, a sentence in natural language is never interpreted in isolation. It is always part of a context, and its meaning is highly dependent on that context. Our theory should explain the different ways in which the "setting" of a sentence can affect its meaning. It must deal both with the linguistic setting (the context within the discourse) and the real-world setting (the way meaning interacts with knowledge of non-linguistic facts.)

4.1.2 The Semantic System

With definite goals in mind for a semantic system, we can now ask how to implement it. First let us look at what it should know about English. As we have been emphasizing throughout the paper, a language is not a set of abstract symbols. It is a system for conveying meaning, and has evolved with very special mechanisms for conveying just those aspects of meaning needed for human communication.

In section 3.1 we discussed the person's "model of the world" which is organized around notions of "objects", having "properties" and entering into "relationships." In 3.1.3, we combined these to form more complicated logical expressions. If we look at the properties of English syntax (as described in section 2.3) we see that these basic elements of the "world model" are just what English is good at conveying.

For describing objects, we have the NOUN GROUP. It contains a noun, which indicates the kind of object being talked about; adjectives and classifiers, which describe properties of the object; and a complex system of quantifiers and determiners to tell us its logical status -- whether we are discussing a particular object, ("the sun"), a class of objects ("people"), a particular set of objects ("John's lizards"), an unspecified set containing a specified number of objects ("three bananas"), etc. The details (described in section 4.2) are complex, but the important thing is the existence of a systematic structure which we can analyze.

For describing relationships and events, we have the CLAUSE, PREPOSITION GROUP, and ADJECTIVE GROUP. The CLAUSE is especially suited for dealing with relationships having a particular time reference, working in coordination with the VERB GROUP, whose main function is to convey information about time, using an ingenious system of tenses. Clauses can also be used to represent an event or relationship as an object (as in "His going pleased me."), or to use relationships to modify a particular object within a NOUN GROUP (in "the man who broke the bank"). PREPOSITION groups are a less flexible but simpler way of expressing relationships which do not need modifiers such as time,

place, and manner (such as "the man in the blue vest"). ADJECTIVE GROUPS are used in some constructions to describe properties and some special kinds of relationships of objects (such as "Her gift was bigger than a breadbox.")

The semantic system is built around a group of about a dozen programs which are experts at looking at these particular syntactic structures. They look at both the structures and the meanings of the words to build up PLANNER theorems which will be used by the deductive mechanism. It is important to remember that the parser uses systemic grammar so the semantic programs can look directly for features such as PASSIVE or PLURAL or QUESTION to make decisions about the meaning of the sentence or phrase.

Since each of these semantic "specialists" can work separately, there is no need to wait for a complete parsing before beginning semantic analysis. The NOUN GROUP specialist can be called as soon as a NOUN GROUP has been parsed, to see whether it makes sense before the parser goes on. In fact, the task can be broken up, and a preliminary NOUN GROUP specialist can be called in the middle of parsing (for example, after finding the noun and adjectives, but before looking for modifying clauses or prepositional phrases) to see whether it is worth continuing, or whether the supposed combination of adjectives and noun is

nonsensical. This is easy to do, since the grammar is in the form of a program. It is just as easy to call a semantic routine at any time as a syntactic one. Any semantic program also has full power to use the deductive system, and can even call the grammar back to do a special bit of parsing before going on with the semantic analysis. For this reason it is very hard to classify our semantic analysis as "top-down" or "bottom-up". In general we try to analyze each piece of the structure as it is parsed, which is a bottom-up approach. However whenever there is a reason to delay a part of the analysis until some of the larger structure has been analyzed, it is just as easy to write the semantic specialist programs in this top-down manner. In fact in our system both approaches are found in some of the programs.

4.1.3 Words

Our system needs to deal with two different kinds of words. Some words have to be included in the general knowledge of the English language. Words like "that" or "than", in "He knew that they were madder than hornets." would be difficult to define except in terms of their place in the sentence structure. They are being used as signals of certain syntactic structures and features, and have no meaning except for this signalling (which is recognized by

the grammar). These are often called "function words" in distinction to the "content words" which make up the bulk of our vocabulary. This is not a sharp distinction, since many words serve a combination of purposes (for example, numbers are basically "function words", but each one has its unique meaning). However we can generally distinguish between words like "that" and "than" whose meanings are built into the system, and words like "snake", "under", and "walk", which surely are not.

The definitions of "content words" should not have to include "expert" knowledge about the semantics or grammar of the language. In defining the word "mighty", we should not have to worry about whether it appears in "The sword is mighty," or "'the mightiest warrior", or "a man mightier than a locomotive." We should be able to say "'mighty' means having the property which we represent conceptually as #MIGHT", and let the semantic system do the rest. Similarly, the definition of a verb such as "damage" should deal abstractly with a "semantic subject", without worrying whether it will be in the form "They damaged his reputation.", or "Which box did they damage?" or "the damaged merchandise" or "nothing was damaged by the storm." In each of these cases, the thing doing the damaging appears at a different place in the structure (or is understood),

and some part of the semantic system must find that place.

We need a semantic language for expressing definitions in a way which does not depend on the grammar or the particular semantic programs. Each of our "specialists" which looks at the meanings of words should be able to interpret those statements in the semantic language which might be relevant to its job.

We will see in section 4.2 that there are very simple formats for defining simple verbs, nouns, adjectives, and prepositions, and that in fact these definitions do not look much like programs at all. Why then do we call this a "language" instead of saying that we have a set of special formats for defining words? The distinction becomes important for all of the irregular cases and the idiosyncracies that words can have. For example, if we say "The block is on the roof of the car.", "the roof of the car" is a NG referring to a particular object which is a roof. But if we say "The block is on the right of the box", we are not referring to a particular object which is a "right". The normal noun-group mechanism for describing objects is being used instead to describe a relationship between the block and the box. We could reprogram our NOUN GROUP semantic specialist to recognize this special case and treat it differently, but this is a path leading to a

roadblock. We will not be able to anticipate every case, and as the program becomes more and more patched, it will become harder to change and less likely to work.

What we need is a flexible way of defining words, so that the word "right" itself can cause the right things to happen in semantic interpretation, without changing the system. In our programs, this is achieved by letting the definition of each word be a LISP program to be run at an appropriate time in the semantic analysis. For the simple cases, we have standard functions which use a special format for usual types of definitions. In the complex cases we have a platform from which to operate, doing whatever calculations and changes of the environment needed to have the right effect. This flexibility is important in many places. For example, the word "one" when used as a noun (as in "the green one") has a very special use for referring back to previously mentioned nouns. It could not be defined by a simple format, as could "block" or "dog", since it involves complex decisions about what is really being referred to, and needs access to the previous discourse. In our system, its definition as a noun is compatible with the definitions of all other nouns -- the semantic specialists don't know anything about it. When the NOUN GROUP specialist is ready to use the definition of the noun, it

calls it as a program. In the usual case, this program sets up a standard data structure. In the case of "one", it calls a heuristic program for understanding back-references, and its effect on the meaning will depend on the discourse. Similarly, the verb "be" is called like any other verb by the semantic specialist, but in fact its definition is a complex program describing its many different uses.

4.1.4 Discourse

At the beginning of our discussion of semantics, we described why a semantic system should deal with the effect of "setting" on the meaning of a sentence. A semantic theory can account for three different types of context.

First, there is the local discourse context, which covers the discourse immediately preceding the sentence, and is important to semantic mechanisms like pronoun reference. If we ask the question "Did you put it on a green one?" or "Why?" or "How many of them were there then?", we assume that it will be possible to fill in the missing information from the immediate discourse. There are a number of special mechanisms for using this kind of information, and they form part of a semantic theory.

Second, there is an overall discourse context. If we say "The group didn't have an identity.", the hearer will interpret it very differently depending on whether we are

discussing mathematics or sociology. In addition to the effects of general subject on choosing between meanings of a word, there is an effect of the context of the particular things being discussed. If we are talking about Argentina, and we say "The government is corrupt.", then it is clear that we mean the government of Argentina. If we say "Pick up the pyramid.", and there are three pyramids on the table, it will not be clear which one is meant. But if this immediately follows the statement "There is a block and a pyramid in the box.", then the reference is clearly to the pyramid in the box. This would have been clear even if there had been several sentences between, discussing the block. Therefore this is a different problem than the local discourse of pronoun reference. A semantic theory must deal with all of these different forms of overall discourse context.

Finally, there is a context of knowledge about the world, and the way that knowledge effects our understanding of language. If we say "The city councilmen refused the demonstrators a permit because they feared violence.", the pronoun "they" will have a different interpretation than if we said "The city councilmen refused the demonstrators a permit because they advocated revolution." We understand this because of our sophisticated knowledge of councilmen,

demonstrators, and politics -- no set of syntactic or semantic rules could interpret this pronoun reference without using knowledge of the world. Of course a semantic theory does not include a theory of political power groups, but it must explain the ways in which this kind of knowledge can interact with linguistic knowledge in interpreting a sentence.

Knowledge of the world may affect not only such things as the interpretation of pronouns, but may alter the parsing of the syntactic structures as well. If we see the sentence "He hit the car with a rock." the structure will be analyzed differently from "He hit the car with a dented fender.", since we know that cars have fenders, but not rocks.

In our system, most of this discourse knowledge is called on by the semantic specialists, and by particular words such as "one", "it", "then", "there", etc. We have concentrated particularly on local discourse context, and the ways in which English carries information from one sentence to the next. A number of special pieces of information are kept, such as the time, place, and objects mentioned in the previous sentence. This information is referenced by special structures and words (for example, pronouns look back for objects, while "then" and "there" look at the time and place references). The meaning of the

entire sentence can be referred to in order to answer a question like "Why did you do that?" or just "Why?".

There are two facilities for handling overall discourse context. The first is a mechanism for assigning a "plausability factor" to an interpretation of a word. For example, the definition of the word "bank" might include the fact that if we are discussing money, it is most likely to mean a financial institution, while if we are discussing rivers, it probably means the edge of the land. Our system allows the definition of a word to include a program to compute a "plausability factor" (an arbitrary additive constant) for each interpretation. This computation might involve looking at the rest of the sentence for key words, or might use some more general idea, like keeping track of the general area of discussion (perhaps in some sort of network or block structure) and letting the plausibility of a particular meaning depend on its "distance" from the current topic. This has not been implemented since we have included only a single topic of discourse in the vocabulary. It is discussed further in section 5.2.

The second type of overall discourse context involves the objects which have been previously mentioned. Whenever an object or one of its properties is mentioned, either by the human or the computer, a note is made of the time.

Later, if we use a phrase like "the pyramid", and the meaning is not clear, the system can look for the one most recently mentioned.

Finally, the knowledge of the world can enter into the semantic interpretation. We have mentioned that the grammar can ask the semantic interpreter "Does this NOUN GROUP make sense?" before continuing the parsing. The semantics programs can in turn call on PLANNER to make any deductions needed to decide on its sensibility. Thus information about the world can guide the parsing directly.

4.1.5 Ambiguity

A semantic theory must have some way to account for multiple meanings of words, phrases, and sentences. We would like to explain not only how multiple interpretations can occur, but also how the hearer sorts them out to pick a single meaning.

As a start, we must allow words to have several "senses", and must be able to have multiple interpretations of phrases and sentences to correspond to them. Next we must realize that the syntactic structures can also lead to semantic ambiguities. The sentence "A man sitting in this room fired the fatal shot." will be ambiguous even if we agree on a single meaning for each word. If spoken by Perry Mason at a dramatic moment in the courtroom, it means "a man

who is sitting in this room", but if spoken by the detectives when they broke into the empty hotel room across the street from the scene of the crime, it means "who was sitting in this room". This could be treated as a syntactic ambiguity in the deep structure, but in our analysis it is instead treated as a semantic ambiguity, since it must be resolved by semantic criteria similar to those for multiple word meanings.

In describing the grammar we pointed out that we do not carry forward simultaneous parsings of a sentence. We try to find the "best" parsing, and try other paths only if we run into trouble. In semantics we take the other approach. If a word has two meanings, then we will build two semantic descriptions simultaneously, and use them to form two separate phrase interpretations.

We can immediately see a problem here. We are in dire danger of a combinatorial explosion. If words A, B, C, and D each have three meanings, then a sentence containing all of them may have $3 \times 3 \times 3 \times 3$, or 81 interpretations. The possibilities for a long sentence are astronomical.

Of course a person does not build up such a tremendous list. As he hears a sentence, he "filters out" all but the most reasonable interpretations. We know that a "ball" can be either a spherical toy or a dancing party, and that

"green" can mean either the color green, or unripe, or inexperienced. But when we see "the green ball", we do not get befuddled with six interpretations, we know that only one makes sense. The use of "green" for "unripe" applies only to fruit, the use as "inexperienced" applies only to people, and the color only to physical objects. The meaning of "ball" as a party fits none of these categories, and the meaning as a "spherical toy" fits only the last one. We can subdivide the world into rough classes such as "animate", "inanimate", "physical", "abstract", "event", "human", etc. and can use this classification scheme to filter out meaningless combinations of interpretations.

Some semantic theories (Katz and Fodor (28)) are based almost completely on this idea. We would like to use it for what it is -- not a complete representation of meaning, but a rough classification which enables us to quickly eliminate fruitless semantic interpretations. Our system has the capacity to use these "semantic markers" to cut down the number of semantic interpretations of any phrase or sentence.

A second method used to reduce the number of different semantic interpretations is to do the interpretation continuously. We do not pile up all possible interpretations of each piece of the sentence, then try to

make logical sense of them together at the end. As each phrase is completed, it is understood. If we come across a phrase like "the colorful ball" in context, we do not keep the two different possible interpretations in mind until the utterance is finished. We immediately look in our memory to see which interpretation is meaningful in our current context of discourse, and use only that meaning in the larger semantic analysis of the sentence. Since our system allows the grammar, semantics and deduction to be easily intermixed, it is possible to do this kind of continuous interpretation.

Finally we must deal with cases where we cannot eliminate all but one meaning as "senseless". There will be sentences where more than one meaning makes sense, and there must be some way to choose the correct one in a given context. In the section on context above, we discussed the use of the overall discourse context in establishing a plausibility factor we can assign to a particular interpretation. By combining the plausibilities of the various parts of a sentence, we can derive an overall factor to help us choose.

There will always be cases where no set of heuristics will be enough. There will be multiple interpretations whose plausibilities will be so close that it would be

simply guessing to choose one. In our sample dialog, we had an example with the word "on". If we say "The block is on top of the pyramid," do we mean "directly on the surface" or "somewhere above"? We might mean either one, and there is no way for the hearer (or computer) to read our minds. The obvious alternative is to ask us to explain more clearly what we mean. As a final resort, our system has the ability to ask questions like "By the word "on" in the phrase "on top of green blocks" did you mean...?". The methods used for handling ambiguity in our system are described in more detail below.

4.1.6 Goals of a Semantic Theory

We have set ourselves very broad goals in our definition of semantics. We have asked for everything which needs to be done, rather than limiting ourselves to those aspects which can be explained and characterized in a neat formalism. How does this compare with the more limited goals of a semantic theory like that of Fodor and Katz (28), which looks only at those aspects of meaning which are independent of the "setting" of a sentence?

We have seen that their theory of "semantic markers" is in fact a part of the "filtering" needed for "exploiting semantic relations in the sentence to eliminate potential ambiguities" ((28)p. 485)", and that the "semantic

distinguishers" are a rudimentary form of the logical descriptions which we build up to describe objects and events. They state that "the distinction between markers and distinguishers is meant to coincide with the distinction between that part of the meaning of a lexical item which is systematic for the language and that part of the meaning of the item which is not." ((28) p. 498). We believe that much more of meaning is systematic, and that a semantic theory can be of a much wider scope.

What about the more restricted goals which a semantic theory might achieve such as "accounting for... the number and content of the readings of a sentence, detecting semantic anomalies, and deciding upon paraphrase relations between sentences."? We see that in a more complete semantic theory, these are not primary goals, but by-products of the analysis. A phrase is a semantic anomaly if the system produces no possible interpretations for it. Two sentences are paraphrases if they produce the same results in semantic analysis, and the "number and content" of the readings of a sentence are the immediate result of its semantic analysis. In addition, we can talk about sentences being anomalies or paraphrases "in context", as well as "without regard to context", since we want the theory to include a systematic analysis of those features of context

which are relevant to understanding. In section 4.2, we give the details of the semantic programs used in our language-understanding system.

4.2 Semantic Structures

In the previous section we outlined the structure of our semantic interpreter, and described the use of semantic "specialists" in analyzing different aspects of linguistic structure. We can look at the function of each specialist as creating a part of a complete description of the meaning of the sentence. We build complex list structures which we will call "semantic structures" to describe objects and relationships. Events are considered to be a certain kind of relationship (involving time), and the class of "object" includes anything which could be treated as an object in English grammar, even if it is as abstract as "truth". There are two basic types of structures used -- one to describe objects, and the other to describe relationships. In general, NOUN GROUPS are interpreted to form object structures, while the other GROUPS and CLAUSES are interpreted to form relationship structures. WORDS already have a structure of their own (their definition) and are used in building up the structures for the larger units which contain them.

4.2.1 Object Semantic Structures

Let us first look at the semantic structures used to describe objects. First, we need the actual PLANNER statements which will be used in deducing things about the

objects. If we have a NG like "a red cube", we can use the formalism described in Chapter 3, to write a description:

```
(THPROG (X1)
  (THGOAL(#IS $?X1 #BLOCK))
  (#EQDIM $?X1)
  (THGOAL(#COLOR $?X1 #RED)))
```

We have arbitrarily selected the variable "X1" to represent the object, and this description says that it should be a block, it should have equal dimensions, and it should be red. (See section 3.4 for the details of the way we represent these and other concepts). We want to have a more complex PLANNER description for a phrase such as "a red cube which supports three pyramids but is not supported by a box". This would be built up from the descriptions for the various objects mentioned, and would end up something like:

```
(THPROG(X1)
  (THGOAL(#IS $?X1 #BLOCK))
  (#EQDIM $?X1)
  (THGOAL(#COLOR $?X1 #RED))
  (THFIND 3 $?X2 (X2) (THGOAL(#IS $?X2 #PYRAMID))
    (THGOAL(#SUPPORT $?X1 $?X2)))
  (THNOT(THPROG(X3)
    (THGOAL(#IS $?X3 #BOX))
    (THGOAL(#SUPPORT $?X3 $?X1))))))
```

We can learn how the semantic specialists work by watching them build the pieces of this structure. First let us take the simpler NG, "a red cube". The first NG specialist doesn't start work until after the noun has been parsed. The PLANNER description is then built backwards,

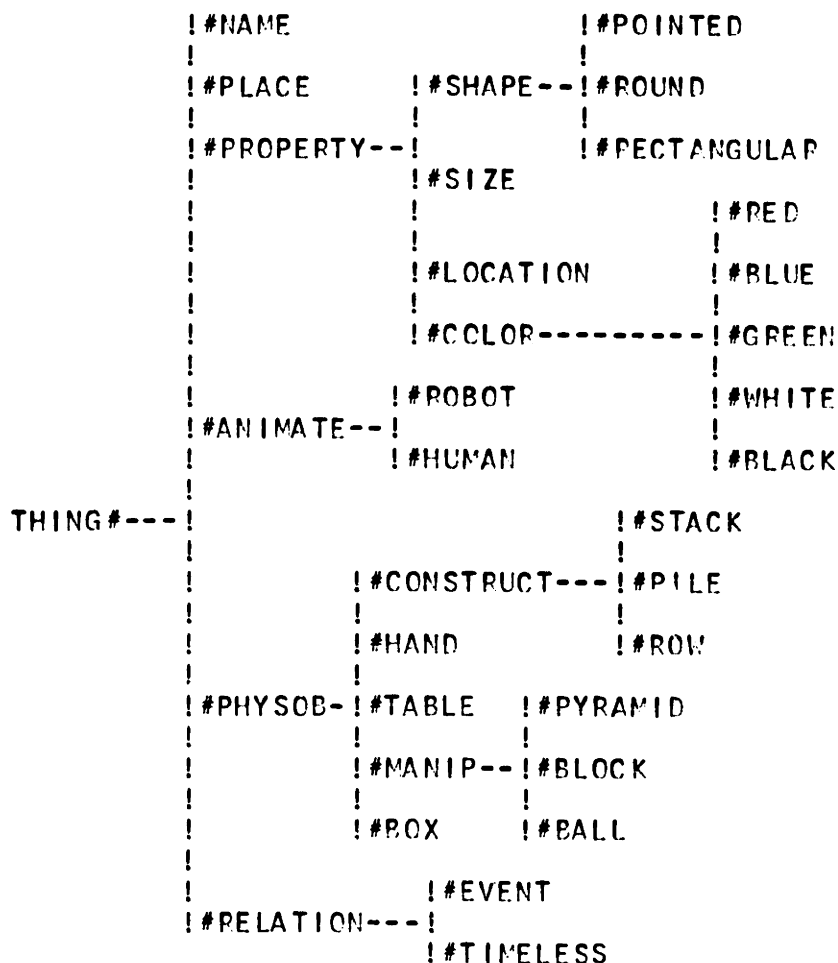
starting with the noun, and continuing in right-to-left order through the classifiers and adjectives. The beginning of the NG, with the determiner, number, and ordinal is handled by a special part of the NG specialist which we will describe later. The first NG specialist is named SMNG1 -- all of the specialists' names begin with SM (for "semantic"), followed by the name of the unit they work with, followed by a number indicating the order in which they are called. SMNG1 sets up an environment (we will describe various parts of it as we go), then calls the definition of the noun. (Remember that definitions are in the form of programs). For words like simple nouns we want to have a standard function to define them easily. What should the definition include? Clearly it must have some way to indicate the PLANNER statements which are the heart of its meaning. We use the symbol "***" to represent the object, so our definition contains the expression:

```
((#IS *** #BLOCK)(#EQDIM ***))
```

The PLANNER functions such as THPROG and THGOAL will be added by the specialists, since we want to keep the definition as simple as possible.

There is one other part of the definition we would like for a noun -- the semantic markers. In the last section we discussed the use of semantic markers to help filter out

meaningless interpretations of a phrase. In order to do this, our definition needs to attach semantic markers to each description. For our vocabulary about the BLOCKS world, we have a tree of semantic markers:



Here we use the same type of diagram we did for our grammars, in which vertical bars represent choices of mutually exclusive markers, while horizontal lines represent

logical dependency. The symbol "#PHYSOB" means "physical object", and "#MANIP" means "manipulable object". The word "cube" then represents an object with the markers (#THING #PHYSOB #MANIP #BLOCK). We shouldn't need to mention all of these in the definition, since the presence of #BLOCK implies the others because of the logical structure of the marker tree.

Our definition of the noun "cube" is then:

```
(NMEANS((#BLOCK)((#IS *** #BLOCK)(#EQDIM ***))))
```

NMEANS is the name of the function for dealing with nouns, and it accepts a list of different meanings for a word. In this case, there is only one meaning. The first thing is the marker list, followed by the reduced PLANNER definition. When NMEANS is executed, it puts this information onto the semantic structure which is being built for the object. It takes care of finding out what markers are implied by the tree, and deciding which predicates need to be in a THGOAL statement (like #IS), and which are LISP predicates (like #EQDIM). We will see later how it also can decide what recommendation lists to put onto the PLANNER goals, to guide the deduction.

SMNG1 then calls the definition for the adjective "red". We would like this definition to include the PLANNER assertion (#COLOR *** #RED), and indicate that it

applies only to physical objects. We can use the exact same format as we did for nouns, defining "red" as:

```
(NMEANS((#PHYSOB)((#COLOR *** #RED)))
```

Notice that there is no distinction made between the use of #PHYSOB here to imply "applies only to physical objects" and the use of #BLOCK in the definition of "cube" to say "this is a block". This is because of the way the markers are implemented. The marker list in a definition is interpreted to mean "this definition applies only if none of the markers here are in conflict with any of the markers already established for the object". Since the noun is the first thing interpreted, its markers cannot possibly conflict, and are simply entered as the initial marker list for the object. The marker programs are designed so that we do not need to limit ourselves to a single tree -- we could classify objects along several dimensions, and set up separate marker trees for each. For example, we might classify objects both by their physical properties (like "hard" and "soft") and by their use.

In adding the meaning of "red" to the semantic structure, the specialist must make a choice in ordering the PLANNER expressions. We remember from section 2.3 that the order of expressions can be important, since variable assignments are done in the order encountered. If we have

the sequence:

```
(THPROG(X)
  (THGOAL(#IS $?X #BLOCK))
  (THGOAL(#COLOR $?X #RED)))
```

PLANNER will look through all of the blocks, checking until it finds one which is red. However if we have:

```
(THPROG(X)
  (THGOAL(#COLOR $?X #RED))
  (THGOAL(#IS $?X #BLOCK)))
```

it will look through all of the red objects until it finds one which is a block. In our tiny world, this isn't of much importance, but if we had a data base which could take phrases like "a man in this room", we would certainly be better off looking around the room first to see what was a man, than looking through all the men in the world to see if one was in the room.

To make this choice we allow each predicate (like #IS or #COLOR) to have associated with it a program which knows how to evaluate its "priority" in any given environment. The program might be as simple as a single number, which would mean "this relation always has this priority". It might on the other hand be a complex heuristic program which takes into account the current state of the world and the discussion. In our program, we have adopted the simpler alternative, assigning fixed priorities in the range 0 to 1000 arbitrarily. By keeping track of the priority of the

expression currently at the top of the PLANNER description, the function NMEANS can decide whether to add a new expression above or below it.

Let us now look at the actual structure which would be built up by our program:

((((X1) 200 (THGOAL(#IS \$?X1 #BLOCK))	PLANNER
(THGOAL(#COLOR \$?X1 #RED))	description
(#EQDIM \$?X1))	
(0 #BLOCK #MANIP #PHYSOB #THING)	markers
(#MANIP #PHYSOB #THING)	systems
X1	variable
(NS INDEF NIL)	determiner
NIL)	ordinal

STRUCTURE 1

Most of the parts of this structure have already been explained. The PLANNER description includes a variable list (we will see its use later), the priority of the first expression, and a list of PLANNER expressions describing the object. The "markers" position lists all of the semantic markers applicable to the object. The 0 at the beginning of the list is the "plausability" of this interpretation. This factor was discussed in section 4.1.4, and is set when we are faced with more than one possible interpretation of a word. Each semantic structure carries along with it its accumulated plausability rating. This will remain 0 unless

it is set specifically by an ambiguity.

The "systems" position is a list of all of the nodes in the set of marker trees (remember that there can be more than one) which have already had a branch selected. It is used in looking for marker conflicts. The "variable" is the variable name chosen to represent this object. The system generates it from the set X1, X2, X3..., providing a new one for each new structure. The only two positions left are the determiner and the ordinal. These are explained in section 4.2.4

4.2.2 Relative Clauses

Let us now take a slightly more complicated NG, "a red cube which supports a pyramid," and follow the parsing and semantic analysis. First, the NG parsing program finds the determiner ("a"), adjective ("red"), and noun ("cube"). At this point SMNG1 is called and creates the structure described in the previous section. Notice that the NG is not finished when SMNG1 is called -- it has only reached a point where we can do a first analysis.

Next the NG program looks for a qualifier, and calls the CLAUSE part of the grammar by (PARSE CLAUSE RSQ). The feature RSQ (rank shifted qualifier) informs the CLAUSE program that it should look for a PELWD like "which". It does, and then looks for a VG, succeeding with "supports".

The VG program calls its own semantic specialist to analyze the time reference of the clause, but we will ignore this for now. Next, since "support" is transitive, the CLAUSE looks for an object, and calls the NG program. This operates in the same way as before, producing a semantic structure to describe "a pyramid". The definition of "pyramid" is:

```
(NMEANS((#PYRAMID)((#IS *** #PYRAMID))))
```

so the resulting structure is:

```
( ( (X2) 200 (THGOAL(#IS $?X2 #PYRAMID)))
  (0 #PYRAMID #MANIP #PHYSOB #THING)
  (#MANIP #PHYSOB #THING))
X2
(NS INDEF NIL)
NIL)
```

STRUCTURE 2

At this point the first CLAUSE specialist is called to analyze the clause "which supports a pyramid". We want to define verbs in a simple way, as we do nouns and adjectives, in which we say something like "if the subject and object are both physical objects, then "support" means the relation #SUPPORT between them in that order". We use the function CMEANS, and can write this formally as:

```
(CMEANS((((#PHYSOB))((#PHYSOB)))(#SUPPORT #1 #2)NIL))
```

All of the extra parentheses are there to leave room for fancier options which will be described later. The

important parts are the semantic marker lists for the objects participating in the relationship, and the actual PLANNEP expression naming it. The symbols #1 and #2 (and #3 if necessary) are used to indicate the objects, and the normal order is 1. semantic subject (SMSUB) 2. semantic first object (SMOB1) 3. semantic second object (SMOB2). Notice that we have prefixed the word "semantic" to each of these. In fact, they may very well not be the actual syntactic subject and objects of the clause, and it is the job of the specialist SMCL1 to find them wherever they are. In this example, the SMSUB is the NG "a red cube" to which the clause is being related. SMCL1 knows this since the parser has noted the feature SURJREL. Before calling the definition of the verb, SMCL1 has found the semantic structure describing "a red cube" and set it as the value of the variable SMSUB. Similarly it has taken the structure for "a pyramid" and put it in SMOB1, since it is the object of the clause. The definition of the verb "support" is now called, and CMEANS uses the information in the definition to build up a relation structure. First it checks to make sure that both objects are compatible with their respective marker lists. The marker lists are in the same order as the symbols #1, #2, and #3.

Next SMCL1 substitutes the objects into the relation.

If we inserted the actual semantic structures, the result would be hard to read and time-consuming to print. Instead, the NG specialists assign a name to each object structure, from the set NG1, NG2, NG3, ... We therefore get (#SUPPORT NG1 NG2) as the description of the relationship. The final semantic structure for the clause (after a second specialist, SMCL2 has had a chance to look for modifiers and rearrange the structure into a convenient form) is:

```
( NG1      (#SUPPORT NG1 NG2)   NIL)  (0))
  rel      relation             neg  markers
```

The position marked "rel" holds the name of the NG description to which this clause serves as a modifier. We will see later that it can be used in a more general way as well. The "relation" is the material for PLANNER to use, and "neg" marks whether the clause is negative or not.

The last element is a set of semantic markers and a priority, just as we had with object descriptions. Relationships have the full capability to use semantic markers just as objects do, and at an early stage of building a relation structure, it contains a PLANNER description, markers, and systems in the identical form to those for object structures (this is to share some of the programs, such as those which check for conflicts between markers). We can classify different types of events and

If we inserted the actual semantic structures, the result would be hard to read and time-consuming to print. Instead, the NG specialists assign a name to each object structure, from the set NG1, NG2, NG3, ... We therefore get (#SUPPORT NG1 NG2) as the description of the relationship. The final semantic structure for the clause (after a second specialist, SMCL2 has had a chance to look for modifiers and rearrange the structure into a convenient form) is:

```
( NG1      (#SUPPORT NG1 NG2)   NIL)  (0)
  rel          relation         neg  markers
```

The position marked "rel" holds the name of the NG description to which this clause serves as a modifier. We will see later that it can be used in a more general way as well. The "relation" is the material for PLANNER to use, and "neg" marks whether the clause is negative or not.

The last element is a set of semantic markers and a priority, just as we had with object descriptions. Relationships have the full capability to use semantic markers just as objects do, and at an early stage of building a relation structure, it contains a PLANNER description, markers, and systems in the identical form to those for object structures (this is to share some of the programs, such as those which check for conflicts between markers). We can classify different types of events and

relationships (for example those which are changeable, those which involve physical motion, etc.) and use the markers to help filter out interpretations of clause modifiers. For example, the modifying PPEPG "without the shopping list" in "He left the house without the shopping list" has a very different interpretation from "without a hammer" in "He built the house without a hammer." He was not going to use the shopping list to help him leave the house! If we had a classification of activities which included those involving motion and those using tools, we could choose the correct interpretation. This will be discussed more in the section on types of PPEPG.

In our limited world, we have not set up a marker tree for relationships and events, so we have not included any markers in the definition of "support". The marker list therefore contains only the plausibility, 0. The "NIL" in the definition indicates that there are no markers, and would be replaced by a list of markers if we wanted to use them.

The clause is now finished, and the specialist on relative clauses (SMRSQ) is called. The problem is to take the information contained in the PLANNER descriptions of the objects involved in the relation, along with the relation itself, and to put it all onto the PLANNER description of

the single object to which the clause is being related. The way in which this is done depends on the exact form of the different objects (particularly on their determiners). In this case, it is relatively easy, and our description of "a red cube which supports a pyramid" becomes:

```
( ( ((X1 X2) 200 (THGOAL(#IS $?X1 #BLOCK))
      (THGOAL(#COLOR $?X1 #RED))
      (#EQDIM $?X1)
      (THGOAL(#IS $?X2 #PYRAMID))
      (THGOAL(#SUPPORT $?X1 $?X2)))

  (0 #BLOCK #MANIP #PHYSOB #THING)
  (#MANIP #PHYSOB #THING))
X1
(NS INDEF NIL)
NIL)
```

STRUCTURE 3

The only thing which has changed is the **PLANNER** description, which now holds all of the necessary information. Its variable list contains both X1 and X2, and these variable names have been substituted for the symbols NG1 and NG2 in the relation. This is now the complete semantic structure for the NG "a red cube which supports a pyramid". In Section 4.2.4 we will describe how a relative clause works with other types of NG descriptions.

4.2.3 Preposition Groups

If we compare the phrase "a red cube which supports a pyramid" with the phrase "a red cube under a pyramid, we see that relative clauses and qualifying prepositional phrases

are very similar in structure and meaning. In fact, they are handled almost identically by the semantic analyzers. The definition of a preposition like "under" uses the same function as the definition of a verb like "support", saying "if the semantic subject and object are both physical objects, then the object is #ABOVE the subject" (Remember that in our BLOCKS world we chose to represent all vertical space relations using the concept #ABOVE). This can be formalized as:

```
(CMEANS((((#PHYSOP))((#PHYSOB)))(#ABOVE #2 #1)NIL)
```

Again, the symbols #1 and #2 refer to the semantic subject and semantic first object, but in the case of a preposition group used as a qualifier, the SMSUB is the NG of which the PREPG is a part, while the SMOB1 is the object of the PREPG (the PREPOBJ). As with clauses, the situation may be more complex. For example, in a sentence like "Who was the man I saw you with last night?", the SMOBJ of the PREP "with" is the question element "who" in the MAJOP CLAUSE. However, the PREPG specialist (SMPREP) takes care of all this, and in defining a preposition, we can deal directly with the SMSUB and the SMOB1. Notice that if we had been defining "above" instead of "under", everything would have been the same except that the relation would have been (#ABOVE #1 #2) instead of (#ABOVE #2 #1). If the PREPG

is an adjunct to a CLAUSE, the SMSUBJ is the relationship structure defining the CLAUSE. The definition of our preposition can then use the semantic markers we have given to relationship descriptions. In the dictionary, definitions of prepositions are kept separated into those for modifying objects and those for modifying relationships.

4.2.4 Types of Object Descriptions

In our examples so far, all of the objects described have been singular and INDEFinite, like "a red cube", and the semantic system has been able to assign them a PLANNEP variable and use it in building their properties into the description. Let us consider another simple case, a DEFINite object, as in "a red block which supports the pyramid". The analysis begins exactly as it did for the earlier case, building a description of "red cube", then one of "pyramid." The "pyramid" description differs from Structure 2 in having DEF in place of INDEF in its determiner. This is noted at the very beginning of the analysis, but has no effect until the entire NG (including any qualifiers) has been parsed. At that time, the second NG specialist SMNG2 looks for definite NGs and tries to determine what they refer to before going on (we have pointed out in various places how this is used to guide the parsing). It takes the PLANNER description which has been

built up, and hands it to PLANNING
expression. The result is a
description. Presumably if the
be referring to a particular object
aware of. If more than one object
there are various discourse heuristics
reference, (see Section 4.3.3). If
failure message is produced and
to try something else.

If SMNG2 is able to find the object
it puts it into the description.
When SMPSO relates the description
"a red cube which supports the top
of this. Instead of building
Structure 3, it builds:

```
((X1) 200 (THGOAL(#IS $?X1  
          (#EQDIM $?X1)  
          (THGOAL(#SUPPORT
```

The object itself is used in
dealing with its description.

What if we had asked about
"three pyramids"? In that case

```
(THFIND 3 $?X2 (X2) (THGOAL  
                  (THGOAL
```

If we had said "a red cube wh

pyramids", a fancier THFIND parame

```
(THFIND (0 3 NIL) $?X2 (X2) (THG
(THG
```

Here, the parameter means "we are find any, but if you find 3, cause to numbers, the S'NG1 and RSQ prog relate descriptions of quantified which supports some pyramid" would original indefinite case. "A red block" would be:

```
(THNOT
  (THPROG (X2) (THGOAL(#IS $?X
    (THGOAL(#SUPPORT
```

For "a red cube which supports eve

```
(THNOT
  (THPROG (X2) (THGOAL(#IS $?X
    (THNOT
      (THGOAL(
```

In other words, "there is no pyramid not support". For the robot, "even know about". This is not a requirement of the way we have set up our semantics done as a convenience, and will be is expanded to be able to discuss well as the specific commands and

We similarly handle the whole types of numbers, using the logical

parameters of PLANNER. The work is actually done in two places. SMNG1 takes the words and syntactic features, and generates the "determiner" which was one of the ingredients of our semantic structure for objects. The determiner contains three parts. First, the number is either 'S (singular, but not with the specific number "one"), 'NPL (plural with no specific number), 'NS-PL (ambiguous between the two, as in "the fish"), or a construction containing an actual arithmetic number. This can either be the number alone, or a combination with ">", "<", or "exactly". Thus the two NGs "at most two days" and "fewer than three days" produce the identical determiner, containing "< 3)". The second element of the determiner is either DEF, INDEF, ALL, NO, or NDET (no determiner at all -- as in "We like sheep.") The third is saved for the question types HOWMANY and WHICH, so it is NIL in a NG which is not a QUEST or PEL.

Other specialists such as SMPSQ and the answering routines use this information to produce PLANNER expressions like the ones described above. In addition, there are special programs for cases like the CF NG. In this case, the PREPOBJ is evaluated as a NG first. Therefore if we had the expression "three of the blocks", we would analyze "the blocks" first, and since it is definite, PLANNER would be called to find out what it refers to. It would return a

list of "the blocks", and the OF specialist would use the PLANNER THAMONG function (which chooses its variable bindings from "among" a given list) to produce an expression like: (THEFIND 3 \$?X1 (X1) (THAMONG X1 (QUOTE(:B1 :B4 :B6 :B7))))

Ordinals are treated specially, along with SUPERlative ADJectives. If we have a NG like "the biggest block which supports a pyramid", it is impossible for SMNG1 to add the meaning of "biggest" to the description in the same way as it would add an expression for "big". The block is "biggest" with respect to a group of objects, and that group is not fully defined until the entire NG has been parsed, including the qualifiers. SMNG1 therefore does a partial analysis of the meaning, looking up the name of the measure that particular adjective refers to, then hangs the result in the last niche of the object semantic structure we described in section 4.2.1. After all has been parsed, SMNG2 finds it there and creates a full logical description. In the case of "the biggest block which supports a pyramid", we would get:

```

((X1 X2 X3 X4 > 200
  (THGOAL(#IS $?X1 #BLOCK))
  (THGOAL(#IS $?X2 #PYRAMID))
  (THGOAL(#SUPPORT $?X1 $?X2))
  (THNOT
    (THAND(THGOAL(#IS $?X3 #BLOCK))
      (THGOAL(#IS $?X4 #PYRAMID))
      (THGOAL(#SUPPORT $?X3 $?X4))
      (THGOAL(#MORE #SIZE $?X3 $?X1))))))

```

A similar type of description is generated for other superlatives and ordinals.

4.2.5 The Meaning of Sentences

So far, we have discussed the semantics of objects and the relationships which are used to describe them in preposition groups and relative clauses. Now we will deal with the overall meaning of a sentence as an utterance -- as a statement, a question, or a command. The sentence is analyzed into a relationship semantic structure, and the system must act on it by responding, taking an action, or storing some knowledge.

First let us look at questions. In describing the grammar of clauses (see section 2.3.3) we pointed out the similarities between questions and relative clauses, and they shared a large part of the system network and the parsing program. They also have much in common on a semantic level. We can look at most questions as being a relative clause to some focus element in the sentence.

If we look at the class of WH questions, this

resemblance is easy to see. First we can take a NGQ question, whose question element is a NG. A question like "Which red cube supports a pyramid?" is very closely related to the NG "a red cube which supports a pyramid. We have found that we can answer such a question by actually going through the steps of relating the clause to the object, and building a description of "a red cube which supports a pyramid." We then take this entire PLANNEP description and put it into a THFIND ALL statement, which is evaluated in PLANNER. The result is a list of objects fitting the description, and is in fact the answer to our question. Of course PLANNER might find several objects or no objects meeting the description. In this case we would need an answer like "none of them" or "two of them". See Section 4.4 for the exact way the responses to questions such as these are generated, depending on the relation between the specific question and the data found. If the question is "how many" instead of "which", the system goes through the identical process, but answers by counting rather than naming the objects found.

No matter what type of NGO we have (there is a tremendous variety -- see section 2.3.3) the same method works. We treat the MAJOR clause as a relative clause to the NG which is the question element, and which we call the

focus. This integrates the relationship intended by the clause into the description of that object. PLANNEP then finds all objects satisfying the expanded description, and the results are used to generate an answer.

Next, we have the OADJ questions, like "when", "why", and "how". In these cases we can think of the focus as being on an event rather than on one element of the relation. If we ask "Why did you pick up a block?", we are referring to an event which was stored in PLANNEP's memory as (#PICKUP E23 :B5) where :B5 is the name of the object picked up, and E23 is the arbitrary name which was assigned to the event (see Section 3.4 for a description of the way such information is stored.) We can ask in PLANNEP:

```
(THFIND ALL $?EVENT ($?EVENT $?X)
  (THGOAL(#PICKUP $?EVENT $?X))
  (THGOAL(#IS $?X #BLOCK)))
```

In other words, "Find all of the events in which you picked up a block." This is clearly the first thing which must be done before we can answer "why". Once it has been done, answering is easy, since PLANNEP will return as the value of THFIND a list of names of such events. On the property list of an event we find the name of the event for which it was called as a subgoal (the "reason"). We need only to describe this in English. Similarly if the question is "when", the property list of the event gives its starting

and ending times. If the question is "why" it takes a little more work, since the subgoal tree is stored with only upward links. But with a little looking around on the EVENTLIST, the system can generate a list of all those goals which had as their reason the one mentioned in the sentence. This concept of a relation as being a sort of object called an "event" is useful in other parts of the semantics as well -- for instance in dealing with embedded clauses as in "the block which I told you to pick up".

"Where" is sometimes handled differently, as it may be either a constituent of a clause, such as a location object (LOBJ) (in "Where did you put it?") or an ADJUNCT (as in "Where did you meet him?"). The first case is handled just like the NG case, making the clause a relative, as if it were "the place where you put it", then asking in PLANEP:

```
(THFIND ALL $?PLACE (PLACE EVENT)
  (THGOAL (#PUT $?EVENT :OBJ $?PLACE)))
```

The ADJUNCT case involves thinking about a special #LOCATION assertion, as in:

```
(THFIND ALL $?PLACE (PLACE EVENT)
  (THGOAL (#MEET $?EVENT :YOU :HIM))
  (THGOAL (#LOCATION $?EVENT $?PLACE)))
```

In this example, we have moved away from the BLOCKS world since we have not yet put any actions into the vocabulary which occur at a specific place without that place being

mentioned in the event, such as #PUT. However the semantic system is perfectly capable of handling such cases.

So far, we have seen that we can answer WH- questions by pretending they are a relative to some object, event, or place, and by adding the relationship to the description of this "focus". Next we look at YES-NO questions, in which there is no "question element" to be used as a focus. It is an interesting fact about English that even in a YES-NO question, there is usually a focus. Consider a simple question like "Does the box contain a block?" Someone might answer "Yes, a red one.", almost as if the question had been "Which block does the box contain?" Notice that "Yes, the box." would not have been at all an appropriate answer. Something about "the box" makes it obvious that it is not the focus. It is not its place as subject or object, since "Is a block in the box?" reverses these roles, but demands the same answer. Clearly it is the fact that "a block" is an INDEFinite NG.

The fact that we say "a block" instead of "the block" indicates that we are not sure of a specific object referred to by the description. Even if we do not inquire about it specifically, the listener knows that the information will be new to us, and possibly of interest since we mentioned the object. In answering "Does the box contain a block?", our

system does the same thing it would do with "How many blocks does the box contain?". It adds the relation to the description of a block, and finds all of the objects meeting this description. Of course the verbal answer is different for the two types of question. In one case, "Yes" is sufficient, while in the other "one" is. But the logical deduction needed to derive it is identical. In fact, our system tries to use this information by replying, "Yes, two of them, a red one and a green one." This may sometimes be verbose, but in fact gives a natural sound to the question-answering. It takes on the "intelligent" character of telling the questioner information he would be interested in knowing, even if he didn't ask for it explicitly.

Once we get into YES-NO questions, it is not always easy to tell what the focus is. Only an INDEF NG which is not embedded in another NG can be the focus, but there may be several of them in a sentence. Sometimes there is no way to choose, but that is rare, as in asking a question, people are usually focusing their attention on a particular object or event. There are a number of devices for indicating the focus. For example, using a quantifier, like "any" or a TPRON like "something" emphasizes the NG more than a simple determiner like "a". In both "Does anything green support a block?", and "Does a block support anything green?", the

phrase "anything green" is the focus. When none of these cues are present, the syntactic function of the NG makes a difference. If we ask "is there a block on a table", then "block" rather than "table" is the focus, since it is the subject while the other is inside a PREPG. Our system contains a heuristic program which takes into account the kind of determiners, number features (singular is more likely than plural), syntactic position, and other such factors in choosing a focus. If it is in fact very difficult to choose in a given case, it is likely that the other speaker will be satisfied with more than one choice.

If we look at sentences in the past tense, we see again that we can have an event as a focus. If we ask, "Did Jesse James rob the stagecoach?", a possible answer, interpreting the event as the focus, is "Yes, three times, yesterday, last week, and a year ago." This is closely parallel to our answers to questions in which the focus is an object.

There are some questions which simply cannot be construed as having a focus, such as a present-tense clause with only definite noun groups. These, however, are even easier to answer, since they can be expressed in the form of a simple set of assertions with no variables. The NG analysis finds the actual objects referred to by a definite NG, and these are used in place of the variable in

relationships. We can therefore answer "yes" or "no" by making a goal of the relationship and letting PLANNEP evaluate it.

Next, we have commands in the form of IMPERATIVE sentences. These are handled somewhat differently. If they contain only indefinite objects, they can be treated in the way mentioned above for questions with no focus. If we say "Pick up the red ball.", we get the relationship (#PICKUP :B7) which can be evaluated directly by putting it in a THGOAL statement:

```
(THGOAL (#PICKUP :B7)(THUSE TC-PICKUP))
```

which will carry out the action.

However, if we say "Pick up a red ball.", the situation is different. We could first use THFIND to find a red ball, then put this object in a simple goal statement as we did with "the red ball". This, however, might be a bad idea. If we try to choose a red ball arbitrarily, we might choose one which is out of reach or which is supporting a tower. The robot might fail or be forced to do a lot of work which it could have avoided with a little thought. What we want is to send the theorem which works on the goal a description rather than an object name, and let the theorem choose the specific object it uses, according to the criteria which best suit it. This is the method we have adopted. Remember

that each object semantic structure has a name like "NG45". Before a clause is related to its objects, these are the symbols used in the relationship. When we analyze "Pick up a red ball", it will actually produce (#PICKUP NG45), where NG45 names a structure describing a red ball. We use this directly as a goal statement, calling a special theorem which knows how to use these descriptions. The theorem calls a theorem named TC-FINDCHOOSE, which uses the description of the object, along with a set of "desirable properties" associated with objects used for trying to achieve the goal. #PICKUP may specify that it would prefer picking up something which doesn't support anything, or which is near the hand's current location. Each theorem can ask for whatever it wants. Of course, it may be impossible to find an object which fits all of the requirements, and the theorem has to be satisfied with what it can get. TC-FINDCHOOSE tries to meet the full specifications first, but if it can't find an object (or enough objects in the case of plural), it gradually releases the restrictions. It must always keep the full requirements of the description input in English in order to carry out the specified command. The robot simply tries to be clever about choosing those objects which fit our description but are also the easiest for it to use.

Finally, we have declarative sentences. We have intentionally not emphasized them, as there are dangers in designing a program to accept information in this way. In Chapter 3, we discussed the complex world-model a person has and explained why we felt that intelligence needed a highly structured and coordinated body of knowledge rather than a set of separate uniform facts or axioms. It is comparatively easy to get a program to add new information of the second type, but very difficult to get it to add the first, since this involves understanding the relationship between the new information and whatever is already there. Therefore, although we have included declarative sentences in our dialog (and they are fully handled in the grammar), we believe that before we start trying to "tell" many things to our program, we need to have a better idea of how knowledge should be structured, and the program should approach learning as a problem solving activity rather than a clerical one. This is discussed further in Section 5.1

In our system we have four different ways in which we can use information in a declarative sentence. The first is a simple word definition facility. If we say "A 'marb' is a red block which is behind a box.", the system recognizes that we are defining a new word. It currently recognizes this by the quote marks, but it could just as easily declare

all unknown words as possible new words to do the same. We have not done this as it would eliminate the feature that the system immediately recognizes typing errors without waiting to begin parsing the sentence.

In a definition, the complement is a noun group with its corresponding object description. We save the PLANNER description from this, and generate a new dictionary entry for the word, defined syntactically as a noun, and with its semantic definition being the program "set the object description to the one we saved earlier." Remember that all definitions are programs, so this one fits in with no problem. When it is called on to build part of the description, it simply puts in the description used to define it. If we talk about "two big marbs", the system will build a description exactly like the one for "two big red blocks which are behind a box." Since only the PLANNER description is saved, the new word can be used with any changes of number, determiner, etc.

The second kind of information the system accepts are simple assertions involving a predicate it knows that it doesn't have complete knowledge about. As we mentioned in Section 3.4, the system has complete data about the physical characteristics of the objects in the scene. We have selected #LIKE as an arbitrary relation about which the

system knows nothing except what it is told in the dialog. If we say "I like you," this produces the assertion (#LIKE :FRIEND :SHRDLU) (the name of the robot is :SHRDLU) which is simply added to the data base. The system also plays a trick with the adjective "nice". Instead of having some concept of #NICE, it assumes that the use of "nice" in describing something is really saying more about the speaker than the object, so the definition of "nice" is

```
(NMEANS((#THING)((#LIKE :FRIEND ***))))
```

If we use an object which isn't definite, as in "I like red blocks.", the system uses the object description to generate a simple PLANNER consequent theorem. It creates a theorem of the form:

```
(THCONSE (X1)
  (#LIKE :FRIEND X1)
    (THGOAL (#IS $?X1 #BLOCK))
    (THGOAL (#COLOR $?X1 #RED)))
```

This theorem says "Whenever you want to prove that the user likes something, you can do it by proving that it is a block and it is red." This is added to the theorem data base, and can be used to answer questions. The system does not separate types of non-definite objects. The results would have been the same if the sentence used "any red block", "every red block", "all red blocks", or (wrongly) "a red block."

It does notice the form "no red blocks" and uses this for the fourth kind of information. It sets up an almost identical theorem, but with a kicker at the end. If we say "I like no red blocks.", it sets up the theorem:

```
(THCONSE (X1)
  (#LIKE :FRIEND X1)
    (THGOAL (#IS $?X1 #BLOCK))
    (THGOAL (#COLOR $?X1 #RED)))
  (THFAIL THGOAL)
```

This theorem is called just like the one above when the system is trying to prove that we like something. But this time, after it finds out that the object is a red block, it does not succeed. Instead, it uses the `PLANNER` function `THFAIL` in a powerful way. It doesn't just cause that theorem to fail, but causes the entire goal to fail, regardless of what other theorems there are. We can also accept a sentence like this with a positive `MG` but a negative clause, as in "I don't like the red block" or "I don't like any red blocks."

In answering a question about liking, the system first tries to succeed using the theorems which have been entered by the dialog. If it succeeds, the answer is "yes". If not, it tries again, using a recommendation which first tries all of the dialog theorems, then tries a special theorem which always succeeds. If one of the negative theorems is applicable, it will get called before the

special one is reached, causing the entire goal to fail, generating the answer "No". If there are no applicable theorems at all, the always-succeed theorem will eventually be called, and its success will cause the response "I don't know" since the system has no relevant information.

4.3 The Semantics of Discourse

In section 3.1, we discussed the different types of context which affect the way a sentence is interpreted. In this section we will describe the specific mechanisms used by our program to include context in its interpretation of language. We have concentrated on the "local discourse context", and the ways in which parts of the meaning of a sentence can be referred to by elements of the next sentence. For example, pronouns like "it" can refer to objects which have been previously mentioned or to an entire event, as in "Why did you do it?". Words like "then" and "there" refer back to a previous time and place, and words like "that" can be used to mean "the one most recently mentioned", as in "Explain that sentence."

In addition to referring back to a particular object, we can refer back to a description in order to avoid repeating it. We can say: "Is there a small grey elephant from Zanzibar next to a big one?" Sometimes instead of using "one" to avoid repeating things, we simply omit part of a phrase or sentence. We can reply to "Would you like a corned-beef sandwich?" with "Bring me two." or we can respond to almost anything with "Why?" In these examples, the second sentence includes by implication a part of the first.

These are not really properly called discourse features, since they can appear just as well in a single sentence. In fact, there are some sentences which would be almost impossible to express without using one of these mechanisms, such as: "Find a block which is bigger than anything which supports it." These mechanisms can be used to refer back to anything mentioned previously, whether in an earlier sentence of the speaker, one of the replies to him, or something occurring earlier in the same utterance.

4.3.1 Pronouns

First we will look at the use of pronouns to refer back to objects. Since our robot does not know any people other than the one conversing with it, it has no trouble with the pronouns "you" and "I" since they always refer to the two objects :SHPDLU and :FRIEND.

When the NG program in the grammar finds a NG consisting of a pronoun, it calls the program which is the definition of that pronoun. The definitions of "it" and "they" use a special heuristic program called SMIT, which looks into the discourse for all of the different things they might refer to, and assigns a kind of plausibility value to each interpretation. If more than one is possible, they are carried along simultaneously through the rest of the sentence, and the ambiguity mechanism decides at the end

which is better, including the last-resort effort of printing out a message asking for clarification. This is in the same format as the message used for other ambiguities, as in sentence 24 of Section 1.3:

I'M NOT SURE WHAT YOU MEAN BY "ON TOP
OF "IN THE PHRASE "ON TOP OF GREEN
CUBES " .

DO YOU MEAN:

- 1 - DIRECTLY ON THE SURFACE
- 2 - ANWHERE ON TOP OF ?

If SMIT finds two different interpretations, and one is chosen because of a higher plausibility, the system types out a message to inform us of the assumption made in choosing one interpretation, as in Sentence 3 of Section 1.3:

BY "IT", I ASSUME YOU MEAN THE BLOCK
WHICH IS TALLER THAN THE ONE I AM
HOLDING.

In our discussion of the analysis of pronouns, we will use "it" as a typical pronoun. In most cases, "they" is treated identically except that it checks for agreement with plural rather than singular. The pronouns "he" and "she" never occur in our limited subject matter, but they would be treated as we handle "it", except that they would make an extra check to see that their reference was in fact animate and of the right sex.

The first thing checked by SMIT is whether "it" has

already appeared in the same sentence. English does not allow us to use the same pronoun to refer to two different objects in the same sentence, so we know we must adopt the same interpretation we did the first time. If there were several possible interpretations, the system is careful not to match up one interpretation from one occurrence of "it" with a different one from another occurrence in building an overall interpretation of the sentence.

Similarly, if "it" was used in the previous sentence, it is likely that if used again it will refer to the same thing. In either of these cases, SMIT simply adopts the previous interpretation.

Next, we may be inside a complex grammatical construction such as "a block which is bigger than anything which supports it." English uses the reflexive pronouns, like "itself" to refer back to an object in the same sentence. However, if in going from the pronoun to the reference on the parsing tree, it is necessary to pass through another NG node, "it" is used, since "itself" would refer to the intermediate NG. Notice that if we replaced "it" by "itself" in our sentence, it would no longer refer to the block, but to "anything".

SMIT looks for this case and other related ones. However if this situation exists, the program must work very

differently. Ordinarily, when we refer to "it" we have already finished finding the reference of the NG being referred back to, and "it" can adopt this reference. In this case, we have a circle, where "it" is part of the definition of the object it is referring to. The part of the program which does variable binding in relating objects and clauses is able to recognize this, and treat it correctly by using the same variable for "the block" and "it".

The pronoun may also be referring back to an object in an embedded clause appearing earlier in the same clause, as in "Before you pick up the red cube, clear it off." SMIT looks through the sentence for objects in acceptable places to which "it" might refer. If it doesn't find them there, it begins to look at the previous sentence. The pronoun may refer to any object in the sentence, and the meaning will often determine which it is (as in our example about the demonstrators in the Preface). We therefore cannot eliminate any of the possibilities on syntactic grounds, but can only give them different ratings of "plausibility". For example, we have discussed the importance of a "focus" element in a clause. "It" is more likely to refer to the previous focus than to other elements of the clause. Similarly, the subject is a more likely candidate than an object, and both are more likely than a NG appearing

embedded in a PREPG or a secondary clause.

The system keeps a list of all of the objects referred to in the previous sentence, as well as the entire parsing structure. By using PROGMAP'S functions for exploring a parsing tree, SMIT is able to find the syntactic position of all the possible references and to assign each a plausibility, using a fairly arbitrary but hopefully useful set of values (for example we add 200 for the focus element beyond what it would normally have for its position as subject or object). In order to keep the list of the objects in the last sentence, our semantic system has to do a certain amount of extra work. If we ask the question: "Is any block supported by three pyramids?", the PLANNER expression produced is:

```
(THFIND ALL $?X1 (X1)
  (THGOAL(#IS $?X1 #BLOCK))
  (THFIND 3 $?X2 (X2)
    (THGOAL(#IS $?X2 #PYRAMID))
    (THGOAL(#SUPPORT $?X2 $?X1))))
```

Once this is evaluated, it returns a list of all the blocks satisfying the description, but no record of what pyramids supported them. If the next sentence asked "Are they tall?", we would have no objects for "they" to refer to. Special instructions are inserted into our PLANNER descriptions which cause lists like this to be saved.

Finally, "it" can be used in a phrase like "Do it!" to

refer to the entire main event of the last sentence. This LASTEVENT is saved, and SMIT can use it to replace the entire meaning of "do it" with the description generated earlier for the event.

When "that" is used in a phrase like "do that", it is handled in a similar way, but with an interesting difference. If we have the sequence "Why did you pick up the ball?" "To build a stack." "How did you do it?", the phrase "do it" refers to "Pick up a ball". But if we had asked "How did you do that?", it would refer to building a stack. The heuristic is that "that" refers to the event most recently mentioned by anyone, while "it" refers to the event most recently mentioned by the speaker.

In addition to remembering the participants and main event of the previous sentence input, the system also remembers those in its own responses. It also remembers time reference, so the word "then" can refer back to it.

4.3.2 Substitutes and Incompletes

The next group of things the system needs to interpret involve the use of substitute nouns like "one", and incomplete noun groups like "Buy me two." Here we cannot look back for a particular object, but must look for a description. SMIT looks through a list of particular objects for its meaning. SMONE (the program used for "one")

looks back into the input sentence instead, to recover the English description. "One" can be used to stand for part or all of that description.

As with "it", "one" can refer back to something in a previous sentence, the previous reply, or earlier in the same sentence. Here though, there are no restrictions about where in the parsing tree the description can lie. "One" depends more on surface characteristics than on structural differences. For example, it cannot refer back to a NG which is a pronoun or uses a TPRON like "anything". Our program for "one" is not as complete as the one for "it". It is primarily based on the heuristic of "contrast". We often use "one" when we want to contrast two characteristics of basically similar objects, for example "the big red block and the little one." Our system must understand these contrasts to interpret the description properly. We realize that "the little one" means "the little red block", not "the little big red block" or "the little block". In order to do this, our system has as part of its semantic knowledge a list of contrasting adjectives. This information is used not only to decide how much of the description is to be borrowed by "one", but also to decide which description in a sentence "one" is referring to. If we say "The green block supports the big pyramid but not the little one." it is

fairly clear that "one" refers to "pyramid". But if we say "The big block supports the green pyramid but not the little one.", then "one" refers to "block". The only thing different is the change of adjectives -- "big" and "little" contrast, but "green" and "little" do not. Our program looks for such contrasts, and if it finds one, it assumes the most recent contrasting description is the reference. If there is no contrast between the phrase being analyzed and any NG in the same sentence, previous answer, or previous sentence, it then looks for the most recent NG which contains a noun.

It is interesting to note that SMONE causes the system to parse some of its own output. In order to use the fragment of a NG it finds, SMONE must know which elements it can use (such as noun, adjective, and classifier) and which it does not (such as number and determiner). For the noun groups in previous inputs, the parsing is available, but for the reply, only the actual words are available and it is necessary to construct a simple parsing before understanding the meaning of "one".

An incomplete NG, containing only a number or quantifier is used in much the same way as "one". In fact, if we look at the series "Buy me three." "Buy me two." "Buy me one.", we see that it can be considered to be the

same thing. We can also take the opposite view that an incomplete NG actually has an implied noun of "one". This is the way our program for handling incomplete noun groups is done.

Currently our set of contrasts is stored separately as special properties in the dictionary entries of the adjectives involved. It would be much better to combine this with the semantic marker system, or the actual system of PLANNER programs and concepts.

4.3.3 Overall Discourse Context

We have discussed several ways of using overall discourse context in understanding. We have so far experimented with only one of these -- keeping track of what has been mentioned earlier in the discourse. This is not the same as looking back in the previous sentence for pronoun references, as it may involve objects several sentences back or not even in the same sentence. If there are many blocks on the table, we can have a conversation: "What is in the box?" "A block and a pyramid." "What is behind it?" "A red block and another box." "What color is the box?" "Green." "Pick up the two blocks."

The phrase "the two blocks" is to be interpreted as a particular pair of blocks, but there may be others in the scene, and nowhere in the dialog were two blocks mentioned

together. The system needs a way to keep track of when things were mentioned, in order to interpret "the" as "the most recently mentioned" in cases like this.

To do so, we use PLANNER'S facility for giving properties to assertions. When we mention a "green block", the semantic system builds a PLANNER description which includes the assertions:

```
(THGOAL(#IS $?X1 #BLOCK)) (THGOAL(#COLOR $?X1 #GREEN))
```

After the sentence containing this phrase has been interpreted, the system goes back to the PLANNER descriptions and marks all of the assertions which were used, by putting the current sentence number on their property lists. This is also done for the assertions used in generating the descriptions of objects in the answer.

When the semantic programs find a definite NG like "the two red blocks", the second NG specialist (SMNG2) uses PLANNER to make a list of all of the objects which fit the description. If there are the right number for the NG, these are listed as the "reference" of the NG, and the interpretation of that NG is done. If there are fewer, SMNG2 makes a note of the English phrase which was used to build the description, and returns a message to the parser that something has gone wrong.

If the parser manages to parse the sentence

differently, all is well. If not, the system assumes that the NG interpretation was the reason for the failure, and the system uses the stored phrase to print out a message "I don't understand what you mean by..."

However, if there are too many objects which match the description, SMNG2 tries to find out which were mentioned most recently. It does this by using PLANNER to recheck the description for the items it found, but this time using only those assertions mentioned in this or the previous sentence. If it finds the right number, these must be the reference of the NG. If it finds too few, it can reiterate the procedure, but using all of the assertions mentioned in the last two sentences. This backward progress continues until at some point either it finds the right number or the number found jumps from below the right number to above it. In this case a message of failure is returned to the parser as before, but a marker is set so that in case the sentence cannot be understood, the message returned is "I don't know which... you mean", as in sentence 2 of Section 1.3:

```
I DON'T UNDERSTAND WHICH PYRAMID  
YOU MEAN.
```

4.4 Generation of Responses

In this section we will describe the way our language-understanding system generates its linguistic responses. This aspect was not emphasized as much in the research as the problem of understanding language, and we have not developed a general theory of discourse generation. We have written a set of programs which produce a kind of behavior which seemed desirable for carrying on a dialog about the robot world. In particular we dealt with four different aspects of discourse production -- patterned responses, appropriate answers to questions, generating descriptions of objects and events, and the production of fluent discourse.

4.4.1 Patterned Responses

The easiest way to get language behavior from a computer program is to include a set of fixed responses which are appropriate for the situations it will encounter. These responses can be made as complex and varied as we want, since they are created by the programmer, and the program only repeats them. Many language systems make heavy use of these fixed responses to achieve "good sounding" output.

In general, these responses are not flexible enough for a true language system, but there are places where they are appropriate -- for example, when the program simply wants to

acknowledge that something has happened, or that it has reached a certain point in its analysis. Our system uses several types of patterned response. First we have a set of simple responses for specific situations. The system responds "ok" when a command is carried out, "I understand" when a declarative sentence is analyzed, "I don't understand" when a sentence cannot be analyzed, and "I can't" when a command cannot be executed.

A slightly more complex type of response involves "filling in the blank" with a phrase borrowed from the input. The simplest example in our system is "Sorry, I don't know the word "...", please type two spaces." The offending word is taken as it was input and inserted in the blank. The "two spaces" are to clear the input buffer of characters typed after the message was sent. Two slightly more complex types of response involve manipulating the determiners of the phrase which was input. If the user types something like "the three green pyramids", and the system cannot figure out what he is referring to, it types "I don't know which three green pyramids you mean." It has simply replaced "the" with "which" before filling the blank. The "I assume" mechanism does the opposite, replacing an indefinite determiner with "the". If we talk about "some green pyramid" or "a green pyramid", then later refer to

that pyramid as "it", the system can notify us of its interpretation of "it" by saying "by 'it' I assume you mean the green pyramid." Here the system has removed the indefinite determiner or quantifier ("a" or "some") and replaced it with "the". It uses knowledge about various kinds of determiners to fill in the pattern reasonably in a number of cases (for example, possessives are left alone, since they are already definite). It can also handle the use of "they" to refer to several objects.

Our most complex "blank-filling" response is the one which handles ambiguity. It uses parts of the input sentence and pieces of the definitions of words stored in the dictionary. In the response to sentence 24 of the sample dialog (section 1.3):

24. how many things are on top of green
cubes?
the system typed:

I'M NOT SURE WHAT YOU MEAN BY "ON TOP
OF "IN THE PHRASE "ON TOP OF GREEN
CUBES " .

DO YOU MEAN:
1 - DIRECTLY ON THE SURFACE
2 - ANYWHERE ON TOP OF ?

The two phrases "on top of" and "on top of green cubes" were taken directly from the input, and the paraphrases "directly on the surface" and "anywhere on top of" were part of the dictionary definition provided for "on". One of the

answer routines looks at the list of places where the interpretations differ, and generates this request for clarification. The response must eliminate at least one possibility, and the program continues generating questions like this one until only one interpretation remains.

4.4.2 Answering Questions

In order to answer questions in discourse, we need to know what types of responses people expect to different types of questions. In our grammar (section 2.3.3) we classified various types of questions, and we can use this classification to select the answers to use. We answer whenever possible with a complete phrase, which provides as much information as possible. No attempt is made to answer using full sentences, since in fact people rarely answer questions with sentences. The natural response to "which block is in the box?" is "the red block", or "the red one", certainly not "The red block is in the box." We have tried to analyze what sort of phrases provide the most informative and natural responses.

The first case is the WH- question, and within that class, the NGO (see section 2.3.2), involving a question NOUN GROUP, such as "which block", "who", "how many pyramids", or "what". We can divide these into three classes -- "how many", "vague", and "specific". If we are asked "What is in

the box?", we can answer "a blue block and a pyramid" without being more specific at identifying them. If instead we are asked "Which block is in the box?" we must use a more specific description like "the large blue cube which supports a pyramid." We need a program which generates English descriptions of particular objects, and it must be able to generate either an INDEFinite description or a DEFINite one. This program will be described in the next section. The use of its results is straightforward for NGQ questions. If the NG is "what", we generate indefinite descriptions of the object or objects. If it is "which...", we generate a definite description. "Who" is never a problem, since the system only knows of two people, "you", and "I". There are also default responses, so that a question like "Which block supports the table?" can be answered "none of them".

HOWMANY questions are answered with the number of appropriate objects, followed by "of them" to make the discourse smoother. For example, the response to dialog sentence 6, "How many blocks are not in the box?", is "four of them."

The next type of question is the OADJ, such as "why", "when", "how", or "where". The only three which have been implemented so far are "why" "when", and "how", but the

others can be done in an analogous fashion. To answer a question using "why", we have the system's memory of the subgoals it used in achieving its goals in manipulating toy objects. If we can decide what event is being referred to in the question, we can see what goal called it as a subgoal. We can then answer by saying "to..." and describing the higher goal in English. If the event was itself a top level goal, it must have been requested as a command, and we can use the fixed response "because you asked me to". We need a program which creates an English description of an event from its PLANNER description like (#PUTON :B3 :TABLE). It must generate phrases which refer to the objects involved, and combine them into a clause of the proper type with the proper tense. This program is also described in the next section.

We can use the same event-describer to answer "how" questions by describing all of the events which were subgoals used in achieving the event mentioned. We say "by...", then list each event in an "ing" form, as in "by picking up a red block and putting it in the box." If the event was itself a lowest-level goal, the system has no way of looking at its own programs for achieving that goal, and answers "I can't analyze how".

"When" questions are answered similarly -- a time is

named by describing the top-level goal which was being carried out at the time, saying "while I was..." and using the "ing" form to describe the event. This is inappropriate if the question refers directly to that goal (We can't answer "When did you build the stack?" with "while I was building the stack"), and in that case we say "before..." and name the top-level goal immediately following in time. If the goal mentioned was just done, the system replies "just now". In addition to the normal responses, the system has a set of fixed responses such as "never", and "I can't explain a non-existent event" to answer questions which demand them.

Finally we come to YES-NO questions which, paradoxically, are the most complicated. It seems that a one word answer is called for, but this is often impossible and rarely the best way to respond. If we ask "Does the block support three pyramids?", and in fact it supports four, what is the correct answer? The system could ask for clarification of the implicit ambiguity between "at least three" and "exactly three", then answer "yes" or "no". But it is more efficient and helpful to answer "four of them", leaving the speaker to interpret his own question. If there were only two pyramids, the system could respond "no", but it would be more informative to say "no, only two

of them". In any of these cases, we might be even more complete, and say something like "four of them: a blue one, two red ones, and a large green one."

There are three parts we can include in the answer -- "yes" or "no", a number, and a description of the objects. Remember that whenever possible a YES-NO question is treated like a NG question by looking for a focus element (an indefinite NG in the question). A question like "Does the pyramid support a block?" is treated logically like the question "Which block does the pyramid support?" , or "How many blocks does the pyramid support?" All three send a THFIND ALL request to PLANNER, asking for all of the objects fitting the description:

```
(THPROG (X)
  (THGOAL (#IS $?X #BLOCK))
  (THGOAL (#SUPPORT :B5 $?X)))
```

where :B5 is the system's internal name for the pyramid being referred to by "the pyramid" (this would be determined in separate calls to PLANNER). In the case of the HOWMANY question we answer with the number of objects found. For the "which" question, we name the objects. In the case of a YES-NO question, we answer with all three kinds of information, saying "yes, two of them: a large red one and the green one."

The first element is "yes" if the answer is clearly yes

(for example if the number is matched exactly, or the number in the original question was indefinite as in this example), "no" if it is clearly no (for example if there are fewer than requested, none at all, or the request was of a form "exactly..." "at least..." "more than..." etc. and was not met), and is omitted if there is a question about its interpretation (as described above).

The second element, the number, is omitted if the number found matches the request (For example, "Are there three blocks?" is not answered redundantly, "yes, three of them: a green one and two large red ones."). The phrase "of them" following the number is changed to "things" if the focus contains a TPRON like "anything", or "something". If the number found is less than that in the focus, it is preceded by "only...", so the answer comes out "no, only two of them:...")

At the end of a response, we put the description of the objects found, unless the request used a special number format such as "exactly...", "at least..." etc. in which case the system assumes the number is more important than the specific objects. We use the object-naming program in its indefinite mode. If the focus originally appeared as the object of a preposition, we repeat that preposition before the description to clarify the answer. Thus, "Is the

pyramid on a block?" is answered "yes, on a large green one." The unknown agent of a passive like "Is it supported?" is implicitly the object of "by", so the answer is "yes, by the table". If a YES-NO question contains no possible focus since all of its NGS are DEFINITE, as in "Does the table support the box?", the system answers simply "yes" or "no".

4.4.3 Naming Objects and Events

The previous section covers all of the different types of questions the system can handle, and the types of phrases it uses in response. We have not yet explained how it names an object or describes an event. This is done with a set of PLANNER and LISP functions which look at the data base and find relevant information about objects. These programs take advantage of the fact that the subject matter is limited, and would need to be made much more general to handle other subjects. Certain features of objects, such as their color and size, are assumed to be the best way to describe them in all contexts.

First we need to know how the object is basically classified. In our BLOCKS world, we have used the concept #IS to represent this, as in (#IS :HAND #HAND), (#IS :P1 #BLOCK), and (#IS #BLUE #COLOR). The naming program for objects first checks for the unique objects in its world,

"I", "you", "the table", "the box", and "the hand". If the object is one of these, these names are used. Next we check to see if it is a color or shape, in which case the English name is simply the concept name without the "#". So the question "What shape is the pyramid?" is answered "pointed" since it has the feature #POINTED. If the the object is not one of these and is not a #BLOCK, #BALL, or a #PYRAMID the program gives up. If it is one of those three, the correct noun is used (including a special check of dimensions to see if a #BLOCK is a "cube"), and a description is built of its color and size. At each stage of building the description, it is checked to see if it refers uniquely to the object being described. If so, the determiner "the" is put on, and the description used without further addition. If there is only one ball in the scene, it will always be referred to as "the ball".

If the description includes color and size, but still fits more than the desired object, the outcome depends on whether we are looking for a specific description or an nonspecific one. If it is nonspecific, the program puts "a" or "an" on the beginning and produces something like "a large green cube". If it is specific, more information is needed. If it supports anything, the program adds the phrase "which supports..." then includes the English

descriptions (indefinite) of all the objects it supports. If the object supports nothing, the program adds "which is to the right of..." and names all of the objects to the left of the desired one. This still may not characterize the object uniquely in some situations, but the system simply assumes that it does. If at any point in the dialog, an object is given a proper name, it is referred to using only the noun and the phrase "named...", as in "the block named superblock."

Naming events is relatively straightforward. With each event type (such as #PUTON or #STACKUP) we associate a small program which generates an English name for the event and combines it properly with the names of the objects involved. For example, the definition for #PUTON is:

```
(APPEND (VBFIX (QUOTE PUT)) OBJ1 (QUOTE (ON)) OBJ2)
```

VBFIX is a program which puts the verb into the right form for the kind of clause needed to answer the question. (for example, -ing for answering "how", or infinitive for answering "why"). It takes into account the changes in spelling involved in adding endings. OBJ1 and OBJ2 are bound by the system to the English names of the objects involved in the event, using the object-naming program described above. APPEND is the LISP function which puts together the four ingredients end to end. We therefore get

descriptions like "putting a large red cube on the table". There is a special check for the order of particles and objects, so that we output "to pick up the small blue pyramid", but "to pick it up" rather than "to pick up it".

4.4.4 Generating Discourse

The previous sections described a generating capability which can produce reasonable English answers to the different types of questions we might ask. But used by themselves, the features described would produce awkward and stilted responses which would at times be incomprehensible. Even though we have mentioned some discourse-like patterns (like "...of them" following a number), we have not yet discussed the real problems of discourse. The system has three basic ways to use discourse devices in producing its own answers. These are much more limited than the range of discourse features it can understand, but they are sufficient to produce a fluent dialog.

The first problem involves lists of objects. Our initial way of naming more than one object is to simply string the descriptions together with commas and "and". We might end up with an answer like "yes, four of them, a large blue block, a small red cube, a small red cube, and a small red cube." To avoid this redundancy, the object-namer looks for identical descriptions and combines them with the

appropriate number to get "a large blue block and three small red cubes". (Note that it also must change the noun to plural).

The next problem is the use of substitute nouns. We would like to respond to "Is there a red cube which supports a pyramid?" by "yes, a large one" instead of "yes, a large red cube". By comparing the English descriptions of the objects we are naming with the wording of the focus in the input sentence, we can omit those nouns and adjectives they share and replace them by "one".

The third problem is more serious, as ignoring it can lead to incomprehensible responses. Consider the answer to question 32 in the dialog ("How did you do it?"). If we did not use the pronoun "it" or the determiner "that", the response would be:

BY PUTTING A LARGE RED BLOCK ON THE TABLE, THEN LETTING GO OF A LARGE RED BLOCK, THEN PUTTING A LARGE GREEN CUBE ON A LARGE RED BLOCK, THEN LETTING GO OF A LARGE GREEN CUBE, THEN PUTTING THE RED CUBE ON A LARGE GREEN CUBE, THEN LETTING GO OF THE RED CUBE.

How many different blocks and cubes are involved? In describing events, we must have some way to indicate that we are referring to the same object more than once. We can do this using "it" and "that", and at the same time can use these words to improve the smoothness of the discourse in other ways. The system has heuristics which lead it to use

"it" to refer to an object in an event it is describing whenever: 1. the same object was called "it" in the question. 2. the object was called "it" in the question preceding the current one, and "it" was not used in the current one. 3. the object has already been named in the current answer, and was the first object in the answer 4. no objects have yet been named in the current answer, and the object was the only one named in the previous answer.

To refer to an object already named in the current answer, other than the first, the program applies the determiner "that" to the appropriate noun, to get a phrase like "by putting a green block on a red cube then putting that cube in the box."

4.4.5 Future Development

The generation of language is a complex subject, wide open to future development. Our current system is just a beginning, and has some major deficiencies. First, we would like to describe an object by using facts which are relevant to the context. In our simple world, we have declared by fiat that color, size, and support relationships are the important facts about an object. We could just have well have used location to get answers like "the block nearest to the back of the table". With a wider range of subjects, we would need much more sophisticated heuristics for deciding

what features of an object will serve best to identify it to the hearer.

Second, we do not have a way to turn an arbitrary PLANNER expression into English. We can handle only specific objects and simple events. There are a number of applications for a more powerful English generator. For example, in case of ambiguity, we shouldn't have to include special paraphrases in the definition. The system should be able to look at the two PLANNER descriptions and describe the difference directly in English.

The system should be able to tell us more about itself and how it does things. If we ask a question like "How do you build stacks?", it should be able to look at its own programs and convert them to an English description like "First I find a space, then I choose blocks, then I put one of the blocks on that space, then..." PLANNER's structure of goals and subgoals is ideal as a subject for this kind of description, and a great deal could be done along this line. In a more speculative vein, the development of discourse generators which could convert an internal logical format into English might lead to computer essay writers, or translators which could understand the material they were working with.

5.1 Teaching, Telling and Learning

One of the most important requirements of a natural language understanding system is generality. It should not be based on special tricks or shortcuts which limit it to one particular subject or a small subset of grammar, but should be expandable to really handle the full diversity of language. In each of the three preceding chapters we have pointed out that many approaches to language understanding are quite limited, and have tried to illustrate the progression within each sub-area towards more general approaches.

5.1.1 Types of Knowledge

In evaluating the flexibility of a system, we must consider the four different levels of knowledge it contains.

First, there is the "hard core" which cannot really be changed without remaking the entire system. This is its "innate capacity" -- the embodiment of its theory of language. At this level we must deal with such questions as whether we should use a top-down transformational parser, a semantic net, or some other approach to the basic analysis of a sentence, or whether we should have special tables of information or a general notation (such as the predicate calculus) for representing information.

The second level of knowledge is the complex knowledge

about the language and the subject being discussed. This would include such things as the grammar of a language, or the conceptual categories into which the speaker divides his model of the world. If we think about the human speaker, this is a type of knowledge which is obviously not innate (since the grammar would be different for English and Chinese and the set of concepts used would differ for talking about toy blocks and talking about love stories). However it is not something which he learns by being told, or which he changes very easily. Over a period of years, he builds up a store of very complex, interrelated knowledge, which serves as a framework for more specific information.

The third level is our storehouse of knowledge about the details of our language and our world. It includes the meanings of words, and most of what we called "complex knowledge" in section 3.1.3. This would include such things as "A house built on sand cannot stand.", "If you want to pick up a block, first clear off its top.", or "Sunspots cause strange weather.". In human terms, this is the knowledge we are continually learning all of our lives, and forms the bulk of what we are taught in school.

Finally, the fourth level is the set of specific facts which are relevant to a discussion. This includes facts such as "Flight 342 leaves Boston at noon.", "The red block

is 3 inches tall.", or "A banana is hanging above the chair.". This is the easiest type to learn, since it does not demand forming any new interrelationships. It is more like putting a new entry into a table or a new simple assertion into a data base. There is no sharp distinction between levels three and four, but within any given system there will usually be two different ways of handling information corresponding to this distinction (see section 2.2). Let us look at the three areas of syntax, inference, and semantics, and see how these different levels of knowledge relate to language understanding programs and the way they can learn.

5.1.2 Syntax

In syntax it is clear that at the top level of knowledge there will be a basic approach to grammar, whether it be transformations, pattern matching, or finite state networks. In addition, there must be some sort of built in system to carry out the parsing.

Some programs (such as the early translation programs) had the grammar built in as an integral part of the system. In order to add new syntactic information it was necessary to dig into the deepest innards of the system and to understand its details. It was recognized quite early that this approach made them inflexible and extremely difficult

to change. The majority of language systems have instead adopted the use of a "syntax-directed" parser. A grammar is described by a series of rules which are applied by a uniform parsing procedure. In handling simple subsets of English, this turns grammar into a third-level type of knowledge. We can add new single rules (for example, adding the fact that verbs can have a modifying adverb) in a way similar to adding words to a vocabulary -- without worrying about the interaction between rules. This simplicity is deceptive, since it depends on the simplicity of context-free grammars for small subsets of natural language. Once we try to account for the complexities of an entire language with something like a systemic or transformational grammar, we must again pay attention to the complex interrelationships between the rules, and the grammar becomes a tangled web into which any new addition must be carefully fitted. For examples of the complexity of current transformational grammars for English, see (Klima (30)). More recent programs which use transformational grammars (Woods (62) Bobrow and Fraser(4) Thorne(55)) recognize the fact that syntax is not really that simple, and adopt a more interrelated representation such as networks.

In our system we have used programs to express the grammar, as explained in chapter 2. This is not a return to

the original first-level representation, since the grammar programs are completely separate from the system itself. One of the arguments for using syntax-directed parsers was that the grammar rules could be expressed in a uniform way which did not depend on the details of the parsing program. Therefore changes could be made more easily and the grammar was expandable. By designing a special language for writing grammars, we can use a representation which is just as general as syntax-rule tables, but which allows greater flexibility in designing a grammar, and relating it to semantics.

How difficult is it to change our grammar? For small changes (like allowing noun groups to contain only a number, as in "Are there any books? I want three.") only one or two additional lines of program would be needed. For a more substantial change (like adding a new type of modifying clause) we might need as many as a dozen small additions to the grammar in different places which would be affected. The first change could be done with little difficulty by anyone with an understanding of PROGRAMMAR and section 2.2. The second would take a deeper understanding of how the grammar is written, but would still involve only a small amount of programming, and of course would not involve changing the basic system at all. The grammar was written

to be fairly complete and with expansion in mind. It seems flexible enough that we will be able to include as much of the the complexity of English as we want.

What is important in terms of learning is that this is level-two knowledge -- it is the type of knowledge which is learned once in a lifetime by a person (or computer program), and should not need any major changes after childhood. Therefore although it must be changeable, we do not need to worry about "quick" learning techniques. If any learning is studied at this level, we must deal in a sophisticated way with the methods used to learn large amounts of complex interrelated material. Those computer programs which have "learned" syntax (Mcconlogue and Simmons (33) Siklossy(47)) have done so by taking such an oversimplified view of syntax that the results hardly have significance for natural language.

At level three of our knowledge of syntax, we have our knowledge of particular words, their grammatical categories and peculiarities. We need to ask, How easy is it to add new words? How much do we have to know about the grammar to increase the vocabulary? In most systems there are a few words (such as "be", "there", or "than") which have complex and unique grammatical behavior. These are built into the grammar initially at level two. The rest of the vocabulary,

like nouns and verbs, can be specified in a simple format. Our system is no exception. To add the words "cat", "purple", and "walk" to the system, we would only need to know the right abbreviations (from section 2.3) to enter in LISP:

```
(DEFLIST WORD (CAT (NOUN NS))(PURPLE(ADJ))(WALK(VB INF
ITRNS)))
```

This says that "cat" is a singular (NS) NOUN, "purple" is an ADJective, and "walk" is the INFinitive form of an InTRAnSitive VerB.

Can we give this information in English? It would be straightforward to add the right terms to the vocabulary and set up simple PLANNER theorems which would allow us to say "'Cat' is a noun." or "'Walk' is an intransitive verb." It would be an interesting project to see how far this could be extended. Some programs have avoided giving dictionary entries to these "open class" words (like verbs, nouns, and adjectives) and let the parser determine their part of speech from context. (Thorne (55)) This approach is not generally meaningful for a complete language understanding system, since we need a dictionary of meanings. It could be used when adding new words to the system, and could be done so trivially in our input programs, by assigning all unknown words to have all possible "open class" grammatical

features, then letting the parser choose the correct ones for the context.

5.1.3 Inference

In the domain of inference, there has been tremendous variation in how different systems treat knowledge. In the early programs, all of the complex information was at level one (built into the system), while the specific facts were at level four. As we have discussed, this made it very hard to modify or expand the complex information held by the system. In the theorem provers, all of the complex information was treated at the fourth level -- as a set of individual formulas which were treated as isolated facts. At level one, they have a uniform proof procedure as the heart of the system. We have discussed how this lack of information at other levels (information about the interrelationships between different theorems) severely limits this approach. In our system, only simple assertions (such as "Noah is the father of Jafeth.", or "Parent-of is the converse of Child-of.") are dealt with at the lowest level. The rest of the knowledge is in the form of PLANNER theorems which have the ability to include information about their connections to other theorems. Some of these, such as the examples in section 3.1.3, are at the third level, since they are not interwoven into complex relationships with

other parts of the knowledge. Other theorems, such as the BLOCKS programs (section 3.4) for keeping track of a table full of objects, are at level two.

Again we can ask, how easy is it to add or change information at each of the levels. At the two ends, the answer is clear. At the top we have PLANNER and our commitment to its kind of theorem-proving procedures. Any change in this is a major overhaul. At the bottom level, we have simple facts like "The red pyramid is supported by the green cube." These are the facts which the system plays with whenever it is conversing. They can be changed by simply telling information (either in English or PLANNER), and are changed automatically when things happen in the world (for example if we move the red pyramid). The middle levels form the much more interesting problem.

At the second level we have our basic conceptual model of the world. This includes our choice of categories for objects, ways of representing actions, time, place, etc. One of the benefits of PLANNER (and of LISP, in which it is embedded) is that we have a variety of useful facilities to represent our world efficiently. Section 3.4 described the BLOCKS world, and it should be similarly easy to define new worlds of discourse for the system (see below for examples).

The third level presents the most interesting problems

for adding new information to the system. It is simple to do so in PLANNER by adding new theorems, but we would like to do it in English as well. Of the previous systems, the only ones which could accept complex information in English were the theorem provers which dealt with it at the fourth level (as a set of unrelated formulas). In our sample dialog, we have some examples of telling the system simple and slightly complex information in English. Saying "I like blocks which are not red, but I don't like anything which supports a pyramid." created two theorems. The first says, "If you want to prove I like something, prove that it is a block and that it is not red." This is no different from a formula for any theorem prover, since it is not related to the system in any complex way. The second theorem says, "If you are trying to prove that I like something, and you can prove that it supports a pyramid, then give up." This interacts with the other goals and theorems, but in a very specialized way.

Much smarter programs could be built to accept complex information and use it to actually modify the PLANNER theorems already in the data base. For example, we might have a theorem to pick up a block, but it fails whenever the block has something on top of it. We would like to say in English, "When you want to pick up a block, first take

everything off of it.", and have the system add this information to the theorem in the form of an additional goal statement at the beginning. In order to do this, the system must have not only a model of the world it talks about, but also a model of its own behavior, so that it can treat its own programs as data to be manipulated and modified. This is one of the most fascinating directions in which the system could be expanded.

Another is the possibility of letting the system learn from experience. This is a complex problem and can be dealt with at many levels. At a simplistic level, we can have it "learn" specific facts. For example, we have a theorem which proves that a block has its top clear (by proving it supports nothing). As the last line of this, we have the PLANNER statement (THASSERT (#CLEARTOP \$?X)), which says that we should add to the data base the assertion that this block is clear. If we then need the fact again, we don't need to repeat the deduction. In a sense the system has "learned" this fact, since it has been added to the data base without being mentioned in the dialog. But in another sense, it hasn't learned any new information, since nothing can be deduced with this fact that couldn't have been done before using the theorem that already existed. A more interesting type of learning would be shown by changing the

PLANNER theorems for accomplishing a goal, depending on what had been achieved in the past. For example, we might have a goal statement with the recommendation (THTBF THTRUE) meaning try anything you can. If the goal is achieved using some particular theorem, we might have the system change the recommendation to suggest trying that theorem first. At a more advanced stage, we would have a heuristic program which tried to figure out why a particular chain of deduction worked or didn't work in a particular case. It would then modify the recommendations to choose the best theorems in whatever environments came up in the future. It might also recognize the need for new theorems in some cases, and actually build them. This is perhaps closest to human learning. It does not involve juggling parameters or adding new isolated bits of information. Instead it involves figuring out "How are my ideas wrong (or right)?" and "How can I change or generalize them?" It involves a kind of "debugging" of ideas, and is a key reason for representing knowledge as procedures.

5.1.4 Semantics

Since semantics is the least understood part of language understanding, it is difficult to find a clear body of "level one" knowledge on which to base a system. Our system has a basic approach to semantics, explained in

chapter 4, but most of the semantic work is done at level two -- the interrelated group of LISP programs for handling particular semantic jobs. At this level we have two separate areas of knowledge. The first is knowledge about the language, and the way it is structured to convey meaning. This includes knowledge such as "In a passive sentence, the syntactic subject is the semantic object.", "A definite noun group refers to a particular object in the world model," or "'It' is more likely to refer to the subject of the previous sentence than the object." This is closely tied to the grammar, and is about as hard to modify as the grammar programs themselves. The other type of level two knowledge is the network of "semantic features" described in section 4.2. This is peculiar to the domain being discussed, and becomes more complex as the range of discussion increases. As we pointed out, this is currently separate from the network of "concepts" used for inference by PLANNER, but the two could be combined. As with level two knowledge in other areas, this is not something to be quickly learned and changed. Our knowledge of how language conveys meaning grows along with our knowledge of its syntactic structure, and is just as seldom modified.

At the third level we have the bulk of semantic information -- the meanings of individual words. This is

the part which must be easy to change and expand. As in most language understanding systems, this knowledge is in the form of separate dictionary entries, so that new words can be added without changing others. The definition of each word is a program in the "semantic language" described in section 4.3, and we gain great flexibility from this program form. The writer of semantic definitions does not have to be concerned with the exact form of the grammar, and if he wants to enter simple words, he can use a standard function to describe them very simply. Most words can be added by using the functions CMEANS and NMEANS, or by using the particular simple semantic form appropriate to the type of word (for example, we would define "thirteen" by ((NUM 13))). If we come across a type of semantic problem or relationship we hadn't anticipated, or which involves relating things in an unusual way, we can write a simple function as the definition of the word to perform the required operations.

We have tried to design our system so that it would be flexible and could be easily adapted to handle other fields of knowledge and to have a large vocabulary. It would be nice to enter new definitions in English instead of having to use the special semantics language. In our sample dialog, the sentence "A "steeple" is a stack which contains

two cubes and a pyramid." produced a new definition for a noun. This is only possible when we can express the definition of the new word in terms of the old words and concepts. It is a bit deceptive for a language understanding system to allow new words to be added so simply. If we wanted to define the word "face", so that we could talk about the faces of blocks, the system would be lacking the basic concepts and relationships necessary to use the new word. This kind of knowledge is at the second level, and we cannot expect to add it through a simple definition. There must be a powerful heuristic program which recognizes the need for a new concept and which relates this concept to the entire model of the world. In this example, it would have to realize that a face is a part of an object, but is not an object itself. This might have varied consequences throughout the model, wherever relations such as "part" are involved.

Thus although our system can accept definitions of some words, it is a worthwhile but untried research project to design programs which will really be able to learn new words in an interesting way. We believe that this will be much easier within the environment of a problem solving language like PLANNER, and that such programs could well be added to our system.

5.2 Directions for Future Research

In our preface we talked about using computers in a new way. We speculated about the day when we will just tell our computer what we want done, and it will understand. This paper has described a small step in that direction. Where is such research leading? What approaches should we take in the future?

We can see three basic directions in which we could extend our system. First, at present it knows only about a tiny simplified subject. Second, most of what it knows has to be programmed, rather than told or taught. Finally, we can't talk to it at all! We have to type our side of the conversation and read the computer's.

The problem of widening the scope of knowledge involves much more than building bigger memories or more efficient lookup methods. If we want the computer to have a large body of knowledge, the information must be highly structured. The critical issue is to understand the kinds of organization needed. One of the reasons that our system is able to handle many aspects of language which were not possible in earlier systems is that it has a deep understanding of the subject it is discussing. There is a whole body of theorems and concepts associated with the words in the vocabulary, and by making use of this knowledge

in its question-answering and action, its language behavior is more like ours. In going to larger areas of discourse we cannot give up this insistence that the computer must know what it is talking about.

We need a way to integrate large amounts of heterogeneous knowledge into a single system which can make use of it. At the same time, we cannot let the system become overburdened and inefficient by insisting on a stifling generality and uniformity. We want the advantages of specialized types of knowledge and structure that can come from limiting the subject to a small area, but at the same time we must have the flexibility that allows knowledge of different types to interact. PLANNER-like languages may be a beginning toward these new kinds of organization.

There are many different approaches which can be taken towards higher organization of knowledge. We may want to think in terms of a "block-structure" of contexts, each of which carries its own special vocabulary and information. We may think of a network, in which we can think of the "distance" between two concepts or words. It might be possible to deal with a set of specialized "subroutines" for dealing with different kinds of situations. Even for something as seemingly simple as childrens' stories, there are tremendous complexities and a well-structured approach

is necessary.

In section 4.1.4 we discussed some of the ways our system could take advantage of this large-scale structure of knowledge. The subject matter would influence the choice of relevant definitions of words and appropriate theorems. This has been explored very little, and there are many possibilities for further research.

The problem of learning is of great interest not only to those working on practical computer systems, but also to psychologists interested in understanding how learning takes place in other intelligent systems, such as people. We need to understand how the amount of knowledge we already have affects the amount and the way we can learn. Working on a natural language program offers several advantages for studying problems of knowledge and learning. Language represents a body of highly complex knowledge, which itself can provide a rich field for learning tasks with a wide range of difficulties. Also, language is the major vehicle through which people learn about the world. In studying the way that a computer could accept new information in natural language, we are studying a key area in learning. We need to understand the ways in which learning depends on the organization of our knowledge. We need to explore in what ways knowing about its own mentality could allow a computer

to really learn. This is perhaps the most interesting possibility for research, and we have discussed it at length in Section 5.1.

Finally we have the problem of speech communication with computers. Again the issue is not one of more efficient hardware, but one of knowledge. Spoken language calls on the listener to fill in a great deal from his own knowledge and understanding. Words, phrases and whole ideas are conveyed by fragments and mumbles which often serve as little more than a clue as to what they intend. The need for a truly vertical system is much greater for speech than for written language. The analysis at even the lowest level depends on whether the result "makes sense." People can communicate under conditions where it is nearly impossible to pick out individual words or sounds without reference to meaning.

In our system we tried to integrate the syntactic, semantic and deductive programs in a flexible way. We allow meaning to guide the direction of the parsing. Our semantic interpretation is guided by logical deduction and a rudimentary model of what the speaker knows. For spoken language this must be expanded. Perhaps we might look for fragments of sentences and use their meaning to help piece together the rest. Or possibly we could create a unified

really learn. This is perhaps the most interesting possibility for research, and we have discussed it at length in section 5.1.

Finally we have the problem of speech communication on computers. Again the issue is not one of more efficient hardware, but one of knowledge. Spoken language is on the listener to fill in a great deal from his own knowledge and understanding. Words, phrases and whole ideas are conveyed by fragments and mumbles which often serve as little more than a clue as to what they intend. The need for a truly vertical system is much greater for speech than for written language. The analysis at even the lowest level depends on whether the result "makes sense." People can communicate under conditions where it is nearly impossible to pick out individual words or sounds without reference to the context.

In our system we tried to integrate the syntactic, semantic and deductive programs in a flexible way. We allow the user to guide the direction of the parsing. Our semantic interpretation is guided by logical deduction and a preliminary model of what the speaker knows. For spoken language this must be expanded. Perhaps we might look for fragments of sentences and use their meaning to help piece together the rest. Or possibly we could create a unified

system in which the deductive portion could look at the context and propose on the basis of meaning, and the audible clues in the utterance, what it thought the speaker might be saying. It might be possible to have a more multi-dimensional analysis in which prosodic features such as voice intonation could be used to recognize important features of the utterance. This is not at all saying that we should throw syntax overboard in favor of some sort of vague relational structure. Often the most important clues about what is being said are the syntactic clues. What is needed is a grammar which can look for and analyze the different types of important patterns rather than getting tremendously involved with the exact details of structure. Systemic grammar is a step in this direction, and the use of programs for grammars gives the kind of flexibility which would be needed for doing this kind of analysis. It is not clear whether our system in its present form could be adapted to handle spoken language, but its general structure and the basic principles of its operation might well be used.

The challenge of programming a computer to use language is really the challenge of producing intelligence. Thought and language are so closely interwoven that the future of our research in natural language and computers will be

neither a study of linguistic principles, nor a study of "artificial" intelligence, but rather an inquiry into the nature of intelligence itself.

BIBLIOGRAPHY

1. Black, F., "A Deductive Question Answering System," in Minsky (ed.) SEMANTIC INFORMATION PROCESSING, pp. 354-402.
2. Bobrow, Daniel G., "Natural Language Input for a Computer Problem Solving System," in Minsky (ed.), SEMANTIC INFORMATION PROCESSING, pp. 135-215.
3. Bobrow, Daniel, "Syntactic Theory in Computer Implementations," in Borko (ed.), AUTOMATED LANGUAGE PROCESSING, pp. 217-252.
4. Bobrow, Daniel, and J.B. Fraser, "An Augmented State Transition Network Analysis Procedure," Proc. of IJCAI, 1969, pp. 557-568.
5. Borko, Harold (ed.), AUTOMATED LANGUAGE PROCESSING, John Wiley and Sons, New York, 1967.
6. Charniak, Eugene, "Computer Solution of Calculus Word Problems," Proc. of IJCAI, 1969, pp. 303-316.
7. Chomsky, Noam, ASPECTS OF THE THEORY OF SYNTAX, M.I.T. Press, Cambridge, Mass., 1965.
8. Chomsky, Noam, SYNTACTIC STRUCTURES, Mouton and Co., The Hague, 1957.
9. Coles, L. Stephen, "Syntax Directed Interpretation of Natural Language," Doctoral Dissertation, Carnegie Mellon University, 1967.
10. Coles, L. Stephen, "An On-Line Question-Answering System with Natural Language and Pictorial Input," Proc. National ACM Conference, 1968, pp. 157-167.
11. Craig, J.A., S.C. Berezner, H.C. Carney, and C.R. Longyear, "DEACON: Direct English Access and CONTROL," Proc. FJCC 1966, pp. 365-380.
12. Darlington, J., "Translating Ordinary Language into Symbolic Logic," Memo MAC-M-149 Project MAC, M.I.T., 1964.

13. Earley, J.C., "Generating a Recognizer for a BNF Grammar," Comp Center Paper, Carnegie Mellon Univ., 1966.
14. Feigenbaum, Edward A., and J. Feldman, COMPUTERS AND THOUGHT, McGraw-Hill, New York, 1963.
15. Fodor, J.A., and J.J. Katz, (ed.) THE STRUCTURE OF LANGUAGE, Prentice Hall, Englewood Cliffs, N.J., 1964.
16. Garvin, P.L., et. al., "A Syntactic Analyzer Study -- Final Report," Bunker-Ramo Corp., Rome Air Development Center, RADC-TT-65-309, Dec. 1965.
17. Green, Cordell, "Application of Theorem Proving to Problem Solving," Proc. of IJCAI, 1969, pp. 219-240.
18. Green, Cordell, and B. Raphael, "The Use of Theorem-Proving Techniques in Question-Answering Systems," Proc. of ACM National Conference, 1968, pp. 169-181.
19. Green, P.F., A.K. Wolf, C. Chomsky, and K. Laugherty, "BASEBALL: An Automatic Question Answerer," in Feigenbaum and Feldman (ed.) COMPUTERS AND THOUGHT, pp. 207-216.
20. Halliday, M.A.K., "Categories of the Theory of Grammar," WORD 17, 1961.
21. Halliday, M.A.K., "Some Notes on 'Deep' Grammar," JOURNAL OF LINGUISTICS 2, 1966.
22. Halliday, M.A.K., "Notes on Transitivity and Theme in English," JOURNAL OF LINGUISTICS 3, 1967.
23. Hewitt, Carl, PLANNER, MAC-M-386, Project MAC, M.I.T. October, 1968, revised August, 1970.
24. Hewitt, Carl, "PLANNER: A Language for Proving Theorems in Robots," Proc. of IJCAI, 1969, pp. 295-301.
25. Hewitt, Carl, Doctoral Dissertation, M.I.T., forthcoming.
26. Huddleston, R.D., "Rank and Depth," LANGUAGE 41, 1965.

27. Hudson, R.A., "Constituency in a Systemic Description of the English Clause," LINGUA 17, 1967.
28. Katz, J.J., and J.A. Fodor, "The Structure of a Semantic Theory," in Fodor and Katz (ed.), THE STRUCTURE OF LANGUAGE, pp. 479-518.
29. Kellogg, C., "A Natural Language Compiler for On-line Data Management," Proc. of FJCC, 1968, pp. 473-492.
30. Klima, Edward S., "Negation in English," in Fodor and Katz (ed.), THE STRUCTURE OF LANGUAGE, pp. 246-323.
31. Kuno, S. "The Predictive Analyzer and a Path Elimination Technique," CACM 8:7 (July 1965), pp. 453-462.
32. Lindsay, Robert, "Inferential Memory as the Basis of Machines Which Understand Natural Language," in Feigenbaum and Feldman (ed.) COMPUTERS AND THOUGHT, pp. 217-236.
33. McConlogue, K.L., and R. Simmons, "Analyzing English Syntax with a Pattern-Learning Parser," CACM 8:11 (November 1965), pp. 687-698.
34. Michie, D. (ed.), MACHINE INTELLIGENCE 3., American Elsevier Press, New York, 1968.
35. Miller, George, LANGUAGE AND COMMUNICATION, McGraw-Hill, New York, 1951.
36. Minsky, Marvin, (ed.) SEMANTIC INFORMATION PROCESSING, M.I.T. Press, Cambridge, Mass., 1968.
37. Minsky, Marvin, "Matter, Mind, and Models," in Minsky (ed.) SEMANTIC INFORMATION PROCESSING, pp. 425-432.
38. Minsky, Marvin, "Turing Lecture," JACM, Jan., 1970.
39. National Academy of Sciences, LANGUAGE AND MACHINES: COMPUTERS IN TRANSLATION AND LINGUISTICS, National Academy of Sciences, Washington D.C., 1966.
40. Petrick, S., "A Recognition Procedure for Transformational Grammars," Doctoral Dissertation, M.I.T., 1965.

41. Quillian, M. Ross, "Semantic Memory," in Minsky (ed.) SEMANTIC INFORMATION PROCESSING, pp. 216-270.
42. Quillian, M. Ross, "The Teachable Language Comprehender," CACM 12:8 (August 1969), pp. 459-475.
43. Raphael, Bertram, "A Computer Program Which 'Understands'," Proc. of FJCC, 1964, pp. 577-589.
44. Raphael, Bertram, "SIR: A Computer Program for Semantic Information Retrieval," in Minsky (ed.) SEMANTIC INFORMATION PROCESSING, pp. 33-134.
45. Robinson, J.A., "A Machine-Oriented Logic Based on the Resolution Principle," JACM, 12:4 (October 1965), pp. 536-541.
46. Shapiro, Stuart C., and G.H. Woodmansee, "A Net Structure Based Relational Question Answerer: Description and Examples," Proc. of IJCAI, 1969, pp. 325-346.
47. Siklossy, L., "Natural Language Learning by Computer," Doctoral Dissertation, Carnegie Mellon Univ., 1968.
48. Simmons, R.F., J.F. Burger, and R.E. Long, "An Approach Toward Answering English Questions from Text," Proc. of FJCC 1966, pp. 357-363.
49. Simmons, R.F., J.F. Burger, and R. Schwarcz, "A Computational Model of Verbal Understanding," Proc. of FJCC, 1968, pp. 441-456.
50. Slagle, James R., "Experiments with a Deductive Question-Answering Program," CACM, 8:12 (December 1965), pp. 792-798.
51. Sussman, Gerald, T. Winograd, and E. Charniak, "Micro-Planner Reference Manual," AI Memo 203, Project MAC, M.I.T., July, 1970.
52. Tharp, Alan L., and G.K. Krulee, "Using Relational Operators to Structure Long-Term Memory," Proc. of AJCAI, 1969, pp. 579-586.

53. Thompson, F.B., "English for the Computer," Proc. of FJCC, 1966, pp. 349-356.
54. Thorne, J., P. Bratley, and H. Dewar, "The Syntactic Analysis of English by Machine," in Michie, D. (ed.) MACHINE INTELLIGENCE 3., pp. 281-309.
55. Thorne, J., "A Program for the Syntactic Analysis of English Sentences," CACM 12:8 (August 1969), pp. 476-480.
56. Weizenbaum, J., "ELIZA" CACM 9:1 (January 1966), pp. 36-45.
57. Weizenbaum, J., "Contextual Understanding by Computers," CACM 10:8 (August 1967), pp. 474-480.
58. White, Jon L., "Interim LISP Progress Report," AI Memo 190, Project MAC, M.I.T., March, 1970.
59. Winograd, Terry, "Linguistics and the Computer Analysis of Tonal Harmony," JOURNAL OF MUSIC THEORY 12:1, 1968, pp. 2-49.
60. Winograd, Terry, "An Interpretive Theory of Language", unpublished term paper, M.I.T., 1969.
61. Woods, William A., "Semantics for a Question-Answering System," Report No. NSF-19, Aiken Computation Laboratory, Harvard Univ., Sept., 1967.
62. Woods, William A., "Augmented Transition Networks for Natural Language Analysis," Report No. CS-1, Aiken Computation Laboratory, Harvard Univ., Dec., 1969.
63. Woods, W. "Procedural Semantics for a Question-Answer Machine," Proc. FJCC, 1968, pp. 457-471.
64. Zwicky, A.M., J. Friedman, B.C. Hall, and D.E. Walker, "The MITRE Syntactic Analysis Procedure for Transformational Grammars," Proc. of FJCC, 1965, pp. 317-326.

- Thompson, F.B., "English for the Computer," Proc. of FJCC, 1966, pp. 349-356.
- Thorne, J., P. Bratley, and H. Dewar, "The Syntactic Analysis of English by Machine," in Michie, D. (ed.) MACHINE INTELLIGENCE 3., pp. 281-309.
- Thorne, J., "A Program for the Syntactic Analysis of English Sentences," CACM 12:8 (August 1969), pp. 476-480.
- Weizenbaum, J., "ELIZA" CACM 9:1 (January 1966), pp. 36-45.
- Weizenbaum, J., "Contextual Understanding by Computers," CACM 10:8 (August 1967), pp. 474-480.
- White, Jon L., "Interim LISP Progress Report," AI Memo 190, Project MAC, M.I.T., March, 1970.
- Winograd, Terry, "Linguistics and the Computer Analysis of Tonal Harmony," JOURNAL OF MUSIC THEORY 12:1, 1968, pp. 2-49.
- Winograd, Terry, "An Interpretive Theory of Language", unpublished term paper, M.I.T., 1969.
- Woods, William A., "Semantics for a Question-Answering System," Report No. NSF-19, Aiken Computation Laboratory, Harvard Univ., Sept., 1967.
- Woods, William A., "Augmented Transition Networks for Natural Language Analysis," Report No. CS-1, Aiken Computation Laboratory, Harvard Univ., Dec., 1969.
- Woods, W. "Procedural Semantics for a Question-Answer Machine," Proc. FJCC, 1968, pp. 457-471.
- Zwicky, A.M., J. Friedman, B.C. Hall, and D.E. Walker, "The MITRE Syntactic Analysis Procedure for Transformational Grammars," Proc. of FJCC, 1965, pp. 317-326.